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

Volume 3A: System Programming Guide, Part 1

NOTE: The *Intel® 64 and IA-32 Architectures Software Developer's Manual* consists of ten volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference, A-L*, Order Number 253666; *Instruction Set Reference, M-U*, Order Number 253667; *Instruction Set Reference, V*, Order Number 326018; *Instruction Set Reference*, W-Z, Order Number 334569; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669; *System Programming Guide, Part 3*, Order Number 326019; *System Programming Guide, Part 4*, Order Number 332831; *Model-Specific Registers,* Order Number 335592. Refer to all ten volumes when evaluating your design needs.

Order Number: 253668-087US
March 2025

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

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.

All product plans and roadmaps are subject to change without notice.

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.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not “commercial” names and not intended to function as trademarks.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy and b) code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

CONTENTS

PAGE

CHAPTER 1
ABOUT THIS MANUAL

1.1 OVERVIEW OF THE SYSTEM PROGRAMMING GUIDE 1-1

CHAPTER 2
SYSTEM ARCHITECTURE OVERVIEW

2.1 OVERVIEW OF THE SYSTEM-LEVEL ARCHITECTURE 2-1
2.1.1 Global and Local Descriptor Tables 2-3
2.1.1.1 Global and Local Descriptor Tables in IA-32e Mode 2-4
2.1.2 System Segments, Segment Descriptors, and Gates 2-4
2.1.2.1 Gates in IA-32e Mode 2-4
2.1.3 Task-State Segments and Task Gates 2-5
2.1.3.1 Task-State Segments in IA-32e Mode 2-5
2.1.4 Interrupt and Exception Handling 2-5
2.1.4.1 Interrupt and Exception Handling IA-32e Mode 2-5
2.1.5 Memory Management 2-6
2.1.5.1 Memory Management in IA-32e Mode 2-6
2.1.6 System Registers 2-6
2.1.6.1 System Registers in IA-32e Mode 2-7
2.1.7 Other System Resources 2-7
2.2 MODES OF OPERATION 2-7
2.2.1 Extended Feature Enable Register 2-9
2.3 SYSTEM FLAGS AND FIELDS IN THE EFLAGS REGISTER 2-9
2.3.1 System Flags and Fields in IA-32e Mode 2-11
2.4 MEMORY-MANAGEMENT REGISTERS 2-11
2.4.1 Global Descriptor Table Register (GDTR) 2-12
2.4.2 Local Descriptor Table Register (LDTR) 2-12
2.4.3 IDTR Interrupt Descriptor Table Register 2-12
2.4.4 Task Register (TR) 2-13
2.5 CONTROL REGISTERS 2-13
2.5.1 CPUID Qualification of Control Register Flags 2-20
2.6 EXTENDED CONTROL REGISTERS (INCLUDING XCR0) 2-20
2.7 PROTECTION-KEY RIGHTS REGISTERS (PKRU AND IA32_PKRS) 2-22
2.8 SYSTEM INSTRUCTION SUMMARY 2-23
2.8.1 Loading and Storing System Registers 2-24
2.8.2 Verifying of Access Privileges 2-24
2.8.3 Loading and Storing Debug Registers 2-25
2.8.4 Invalidating Caches and TLBs 2-25
2.8.5 Controlling the Processor 2-26
2.8.6 Reading Performance-Monitoring and Time-Stamp Counters 2-26
2.8.6.1 Reading Counters in 64-Bit Mode 2-27
2.8.7 Reading and Writing Model-Specific Registers 2-27
2.8.7.1 Reading and Writing Model-Specific Registers in 64-Bit Mode 2-27
2.8.8 Enabling Processor Extended States 2-27

CHAPTER 3
PROTECTED-MODE MEMORY MANAGEMENT

3.1 MEMORY MANAGEMENT OVERVIEW 3-1
3.2 USING SEGMENTS 3-2
3.2.1 Basic Flat Model 3-3
3.2.2 Protected Flat Model 3-3
3.2.3 Multi-Segment Model 3-4
3.2.4 Segmentation in IA-32e Mode 3-5
3.2.5 Paging and Segmentation 3-5
3.3 PHYSICAL ADDRESS SPACE 3-6
3.3.1 Intel® 64 Processors and Physical Address Space 3-6
3.4 LOGICAL AND LINEAR ADDRESSES 3-6
3.4.1 Logical Address Translation in IA-32e Mode 3-7

PAGE

3.4.2 Segment Selectors. . . . 3-7
3.4.3 Segment Registers 3-8
3.4.4 Segment Loading Instructions in IA-32e Mode 3-9
3.4.5 Segment Descriptors. . . . 3-9
3.4.5.1 Code- and Data-Segment Descriptor Types. . . . 3-12
3.5 SYSTEM DESCRIPTOR TYPES 3-13
3.5.1 Segment Descriptor Tables. . . . 3-14
3.5.2 Segment Descriptor Tables in IA-32e Mode 3-16

CHAPTER 4
LINEAR-ADDRESS PRE-PROCESSING
4.1 ENABLING AND ENUMERATION. . . . 4-1
4.2 MODE-BASED ACCESSES AND LINEAR-ADDRESS-SPACE PARTITIONING 4-1
4.3 LINEAR-ADDRESS-SPACE SEPARATION (LASS) 4-2
4.3.1 Enumeration and Enabling. . . . 4-2
4.3.2 Operation of Linear-Address-Space Separation (LASS) 4-2
4.4 LINEAR-ADDRESS MASKING 4-3
4.4.1 Enumeration, Enabling, and Configuration 4-3
4.4.2 Treatment of Data Accesses with LAM Active 4-4
4.4.3 Paging Interactions 4-4
4.5 CANONICALITY CHECKING 4-4
4.5.1 Memory Accesses 4-5
4.5.2 Loads of the Instruction Pointer (RIP) 4-5
4.5.3 System Registers Containing Linear Addresses. . . . 4-5
4.5.4 TLB-Invalidation Instructions 4-7

CHAPTER 5
PAGING
5.1 PAGING MODES AND CONTROL BITS 5-1
5.1.1 Four Paging Modes 5-1
5.1.2 Paging-Mode Enabling. . . . 5-3
5.1.3 Paging-Mode Modifiers. . . . 5-4
5.1.4 Enumeration of Paging Features by CPUID 5-5
5.2 HIERARCHICAL PAGING STRUCTURES: AN OVERVIEW. . . . 5-7
5.3 32-BIT PAGING 5-9
5.4 PAE PAGING. . . . 5-14
5.4.1 PDPTE Registers. . . . 5-14
5.4.2 Linear-Address Translation with PAE Paging 5-15
5.5 4-LEVEL PAGING AND 5-LEVEL PAGING 5-20
5.5.1 Ordinary Paging and HLAT Paging 5-20
5.5.2 Use of CR3 with Ordinary 4-Level Paging and 5-Level Paging 5-20
5.5.3 Use of HLATP with HLAT 4-Level Paging and 5-Level Paging. . . . 5-22
5.5.4 Linear-Address Translation with 4-Level Paging and 5-Level Paging. . . . 5-22
5.5.5 Restart of HLAT Paging. . . . 5-33
5.6 ACCESS RIGHTS. . . . 5-33
5.6.1 Determination of Access Rights 5-33
5.6.2 Protection Keys 5-36
5.7 PAGE-FAULT EXCEPTIONS 5-37
5.8 ACCESSED AND DIRTY FLAGS 5-39
5.9 PAGING AND MEMORY TYPING 5-39
5.9.1 Paging and Memory Typing When the PAT is Not Supported (Pentium Pro and Pentium II Processors). . . . 5-40
5.9.2 Paging and Memory Typing When the PAT is Supported (Pentium III and More Recent Processor Families). . . . 5-40
5.9.3 Caching Paging-Related Information about Memory Typing 5-41
5.10 CACHING TRANSLATION INFORMATION. . . . 5-41
5.10.1 Process-Context Identifiers (PCIDs). . . . 5-41
5.10.2 Translation Lookaside Buffers (TLBs). . . . 5-42
5.10.2.1 Page Numbers, Page Frames, and Page Offsets 5-42
5.10.2.2 Caching Translations in TLBs 5-43
5.10.2.3 Details of TLB Use. . . . 5-43
5.10.2.4 Global Pages 5-44
5.10.3 Paging-Structure Caches. . . . 5-44
5.10.3.1 Caches for Paging Structures. . . . 5-44

PAGE

5.10.3.2 Using the Paging-Structure Caches to Translate Linear Addresses 5-47
5.10.3.3 Multiple Cached Entries for a Single Paging-Structure Entry 5-47
5.10.4 Invalidation of TLBs and Paging-Structure Caches 5-48
5.10.4.1 Operations that Invalidate TLBs and Paging-Structure Caches 5-48
5.10.4.2 Recommended Invalidation 5-50
5.10.4.3 Optional Invalidation 5-51
5.10.4.4 Delayed Invalidation 5-52
5.10.5 Propagation of Paging-Structure Changes to Multiple Processors 5-52
5.11 INTERACTIONS WITH VIRTUAL-MACHINE EXTENSIONS (VMX) 5-53
5.11.1 VMX Transitions 5-53
5.11.2 VMX Support for Address Translation 5-54
5.12 USING PAGING FOR VIRTUAL MEMORY 5-54
5.13 MAPPING SEGMENTS TO PAGES 5-54

CHAPTER 6
PROTECTION

6.1 ENABLING AND DISABLING SEGMENT AND PAGE PROTECTION 6-1
6.2 FIELDS AND FLAGS USED FOR SEGMENT-LEVEL AND PAGE-LEVEL PROTECTION 6-2
6.2.1 Code-Segment Descriptor in 64-bit Mode 6-3
6.3 LIMIT CHECKING 6-4
6.3.1 Limit Checking in 64-bit Mode 6-5
6.4 TYPE CHECKING 6-5
6.4.1 Null Segment Selector Checking 6-6
6.4.1.1 NULL Segment Checking in 64-bit Mode 6-6
6.5 PRIVILEGE LEVELS 6-6
6.6 PRIVILEGE LEVEL CHECKING WHEN ACCESSING DATA SEGMENTS 6-8
6.6.1 Accessing Data in Code Segments 6-9
6.7 PRIVILEGE LEVEL CHECKING WHEN LOADING THE SS REGISTER 6-10
6.8 PRIVILEGE LEVEL CHECKING WHEN TRANSFERRING PROGRAM CONTROL BETWEEN CODE SEGMENTS 6-10
6.8.1 Direct Calls or Jumps to Code Segments 6-10
6.8.1.1 Accessing Nonconforming Code Segments 6-11
6.8.1.2 Accessing Conforming Code Segments 6-12
6.8.2 Gate Descriptors 6-13
6.8.3 Call Gates 6-13
6.8.3.1 IA-32e Mode Call Gates 6-14
6.8.4 Accessing a Code Segment Through a Call Gate 6-15
6.8.5 Stack Switching 6-17
6.8.5.1 Stack Switching in 64-bit Mode 6-19
6.8.6 Returning from a Called Procedure 6-20
6.8.7 Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions 6-20
6.8.7.1 SYSENTER and SYSEXIT Instructions in IA-32e Mode 6-21
6.8.8 Fast System Calls in 64-Bit Mode 6-22
6.9 PRIVILEGED INSTRUCTIONS 6-23
6.10 POINTER VALIDATION 6-24
6.10.1 Checking Access Rights (LAR Instruction) 6-24
6.10.2 Checking Read/Write Rights (VERR and VERW Instructions) 6-25
6.10.3 Checking That the Pointer Offset Is Within Limits (LSL Instruction) 6-25
6.10.4 Checking Caller Access Privileges (ARPL Instruction) 6-26
6.10.5 Checking Alignment 6-27
6.11 PAGE-LEVEL PROTECTION 6-27
6.11.1 Page-Protection Flags 6-28
6.11.2 Restricting Addressable Domain 6-28
6.11.3 Page Type 6-28
6.11.4 Combining Protection of Both Levels of Page Tables 6-28
6.11.5 Overrides to Page Protection 6-29
6.12 COMBINING PAGE AND SEGMENT PROTECTION 6-29
6.13 PAGE-LEVEL PROTECTION AND EXECUTE-DISABLE BIT 6-30
6.13.1 Detecting and Enabling the Execute-Disable Capability 6-30
6.13.2 Execute-Disable Page Protection 6-30
6.13.3 Reserved Bit Checking 6-31
6.13.4 Exception Handling 6-32

PAGE

CHAPTER 7
INTERRUPT AND EXCEPTION HANDLING

7.1 INTERRUPT AND EXCEPTION OVERVIEW 7-1
7.2 EXCEPTION AND INTERRUPT VECTORS 7-1
7.3 SOURCES OF INTERRUPTS 7-2
7.3.1 External Interrupts 7-2
7.3.2 Maskable Hardware Interrupts 7-3
7.3.3 Software-Generated Interrupts 7-4
7.4 SOURCES OF EXCEPTIONS 7-4
7.4.1 Program-Error Exceptions 7-4
7.4.2 Software-Generated Exceptions 7-4
7.4.3 Machine-Check Exceptions 7-4
7.5 EXCEPTION CLASSIFICATIONS 7-5
7.6 PROGRAM OR TASK RESTART 7-5
7.7 NONMASKABLE INTERRUPT (NMI) 7-6
7.7.1 Handling Multiple NMIs 7-6
7.8 ENABLING AND DISABLING INTERRUPTS 7-6
7.8.1 Masking Maskable Hardware Interrupts 7-7
7.8.2 Masking Instruction Breakpoints 7-7
7.8.3 Masking Exceptions and Interrupts When Switching Stacks 7-8
7.9 PRIORITIZATION OF CONCURRENT EVENTS 7-8
7.10 INTERRUPT DESCRIPTOR TABLE (IDT) 7-9
7.11 IDT DESCRIPTORS 7-10
7.12 EXCEPTION AND INTERRUPT HANDLING 7-11
7.12.1 Exception- or Interrupt-Handler Procedures 7-12
7.12.1.1 Shadow Stack Usage on Transfers to Interrupt and Exception Handling Routines 7-14
7.12.1.2 Protection of Exception- and Interrupt-Handler Procedures 7-16
7.12.1.3 Flag Usage By Exception- or Interrupt-Handler Procedure 7-17
7.12.2 Interrupt Tasks 7-17
7.13 ERROR CODE 7-18
7.14 EXCEPTION AND INTERRUPT HANDLING IN 64-BIT MODE 7-19
7.14.1 64-Bit Mode IDT 7-19
7.14.2 64-Bit Mode Stack Frame 7-20
7.14.3 IRET in IA-32e Mode 7-21
7.14.4 Stack Switching in IA-32e Mode 7-21
7.14.5 Interrupt Stack Table 7-22
7.15 EXCEPTION AND INTERRUPT REFERENCE 7-23
Interrupt 0—Divide Error Exception (#DE) 7-24
Interrupt 1—Debug Exception (#DB) 7-25
Interrupt 2—NMI Interrupt 7-27
Interrupt 3—Breakpoint Exception (#BP) 7-28
Interrupt 4—Overflow Exception (#OF) 7-29
Interrupt 5—BOUND Range Exceeded Exception (#BR) 7-30
Interrupt 6—Invalid Opcode Exception (#UD) 7-31
Interrupt 7—Device Not Available Exception (#NM) 7-32
Interrupt 8—Double Fault Exception (#DF) 7-33
Interrupt 9—Coprocessor Segment Overrun 7-35
Interrupt 10—Invalid TSS Exception (#TS) 7-36
Interrupt 11—Segment Not Present (#NP) 7-38
Interrupt 12—Stack Fault Exception (#SS) 7-40
Interrupt 13—General Protection Exception (#GP) 7-41
Interrupt 14—Page-Fault Exception (#PF) 7-44
Interrupt 16—x87 FPU Floating-Point Error (#MF) 7-48
Interrupt 17—Alignment Check Exception (#AC) 7-50
Interrupt 18—Machine-Check Exception (#MC) 7-52
Interrupt 19—SIMD Floating-Point Exception (#XM) 7-53
Interrupt 20—Virtualization Exception (#VE) 7-55
Interrupt 21—Control Protection Exception (#CP) 7-56
Interrupts 32 to 255—User Defined Interrupts 7-58

CHAPTER 8
USER INTERRUPTS

8.1 INTRODUCTION. . . . 8-1
8.2 ENUMERATION AND ENABLING. . . . 8-1
8.3 USER-INTERRUPT STATE AND USER-INTERRUPT MSRS 8-1
8.3.1 User-Interrupt State. . . . 8-2
8.3.2 User-Interrupt MSRs. . . . 8-2
8.4 EVALUATION AND DELIVERY OF USER INTERRUPTS. . . . 8-3
8.4.1 User-Interrupt Recognition. . . . 8-3
8.4.2 User-Interrupt Delivery 8-4
8.5 USER-INTERRUPT NOTIFICATION IDENTIFICATION AND PROCESSING. . . . 8-5
8.5.1 User-Interrupt Notification Identification 8-6
8.5.2 User-Interrupt Notification Processing. . . . 8-6
8.6 USER-INTERRUPT INSTRUCTIONS 8-7
8.7 FLEXIBLE UPDATES OF UIF BY UIRET 8-7
8.8 USER IPIS 8-7

CHAPTER 9
TASK MANAGEMENT

9.1 TASK MANAGEMENT OVERVIEW 9-1
9.1.1 Task Structure 9-1
9.1.2 Task State 9-2
9.1.3 Executing a Task 9-2
9.2 TASK MANAGEMENT DATA STRUCTURES. . . . 9-3
9.2.1 Task-State Segment (TSS) 9-3
9.2.2 TSS Descriptor 9-5
9.2.3 TSS Descriptor in 64-bit mode. . . . 9-6
9.2.4 Task Register 9-7
9.2.5 Task-Gate Descriptor 9-8
9.3 TASK SWITCHING. . . . 9-9
9.4 TASK LINKING. . . . 9-15
9.4.1 Use of Busy Flag To Prevent Recursive Task Switching. . . . 9-16
9.4.2 Modifying Task Linkages. . . . 9-16
9.5 TASK ADDRESS SPACE. . . . 9-16
9.5.1 Mapping Tasks to the Linear and Physical Address Spaces 9-17
9.5.2 Task Logical Address Space 9-18
9.6 16-BIT TASK-STATE SEGMENT (TSS) 9-18
9.7 TASK MANAGEMENT IN 64-BIT MODE 9-19

CHAPTER 10
MULTIPLE-PROCESSOR MANAGEMENT

10.1 LOCKED ATOMIC OPERATIONS 10-1
10.1.1 Guaranteed Atomic Operations. . . . 10-2
10.1.2 Bus Locking 10-3
10.1.2.1 Automatic Locking. . . . 10-3
10.1.2.2 Software Controlled Bus Locking 10-4
10.1.2.3 Features to Disable Bus Locks. . . . 10-4
10.1.3 Handling Self- and Cross-Modifying Code. . . . 10-5
10.1.4 Effects of a LOCK Operation on Internal Processor Caches 10-6
10.2 MEMORY ORDERING 10-6
10.2.1 Memory Ordering in the Intel® Pentium® and Intel486™ Processors 10-7
10.2.2 Memory Ordering in P6 and More Recent Processor Families. . . . 10-7
10.2.3 Examples Illustrating the Memory-Ordering Principles 10-9
10.2.3.1 Assumptions, Terminology, and Notation 10-9
10.2.3.2 Neither Loads Nor Stores Are Reordered with Like Operations. . . . 10-10
10.2.3.3 Stores Are Not Reordered With Earlier Loads 10-10
10.2.3.4 Loads May Be Reordered with Earlier Stores to Different Locations 10-11
10.2.3.5 Intra-Processor Forwarding Is Allowed. . . . 10-12
10.2.3.6 Stores Are Transitively Visible 10-12
10.2.3.7 Stores Are Seen in a Consistent Order by Other Processors. . . . 10-13
10.2.3.8 Locked Instructions Have a Total Order 10-13
10.2.3.9 Loads and Stores Are Not Reordered with Locked Instructions. . . . 10-13

PAGE

10.2.4 Fast-String Operation and Out-of-Order Stores 10-14
10.2.4.1 Memory-Ordering Model for String Operations on Write-Back (WB) Memory 10-14
10.2.4.2 Examples Illustrating Memory-Ordering Principles for String Operations 10-15
10.2.5 Strengthening or Weakening the Memory-Ordering Model 10-17
10.3 SERIALIZING INSTRUCTIONS 10-18
10.4 MULTIPLE-PROCESSOR (MP) INITIALIZATION 10-20
10.4.1 BSP and AP Processors 10-20
10.4.2 MP Initialization Protocol Requirements and Restrictions 10-21
10.4.3 MP Initialization Protocol Algorithm for MP Systems 10-21
10.4.4 MP Initialization Example 10-22
10.4.4.1 Typical BSP Initialization Sequence 10-22
10.4.4.2 Typical AP Initialization Sequence 10-24
10.4.5 Identifying Logical Processors in an MP System 10-25
10.5 INTEL® HYPER-THREADING TECHNOLOGY AND INTEL® MULTI-CORE TECHNOLOGY 10-26
10.6 DETECTING HARDWARE MULTI-THREADING SUPPORT AND TOPOLOGY 10-26
10.6.1 Initializing Processors Supporting Intel® Hyper-Threading Technology 10-27
10.6.2 Initializing Multi-Core Processors 10-27
10.6.3 Executing Multiple Threads on an Intel® 64 or IA-32 Processor Supporting Hardware Multi-Threading 10-28
10.6.4 Handling Interrupts on an IA-32 Processor Supporting Hardware Multi-Threading 10-28
10.7 INTEL® HYPER-THREADING TECHNOLOGY ARCHITECTURE 10-28
10.7.1 State of the Logical Processors 10-29
10.7.2 APIC Functionality 10-30
10.7.3 Memory Type Range Registers (MTRR) 10-30
10.7.4 Page Attribute Table (PAT) 10-30
10.7.5 Machine Check Architecture 10-30
10.7.6 Debug Registers and Extensions 10-31
10.7.7 Performance Monitoring Counters 10-31
10.7.8 IA32_MISC_ENABLE MSR 10-31
10.7.9 Memory Ordering 10-31
10.7.10 Serializing Instructions 10-31
10.7.11 Microcode Update Resources 10-31
10.7.12 Self Modifying Code 10-32
10.7.13 Implementation-Specific Intel® HT Technology Facilities 10-32
10.7.13.1 Processor Caches 10-32
10.7.13.2 Processor Translation Lookaside Buffers (TLBs) 10-32
10.7.13.3 Thermal Monitor 10-33
10.7.13.4 External Signal Compatibility 10-33
10.8 MULTI-CORE ARCHITECTURE 10-33
10.8.1 Logical Processor Support 10-34
10.8.2 Memory Type Range Registers (MTRR) 10-34
10.8.3 Performance Monitoring Counters 10-34
10.8.4 IA32_MISC_ENABLE MSR 10-34
10.8.5 Microcode Update Resources 10-34
10.9 PROGRAMMING CONSIDERATIONS FOR HARDWARE MULTI-THREADING CAPABLE PROCESSORS 10-35
10.9.1 Hierarchical Mapping of Shared Resources 10-35
10.9.2 Hierarchical Mapping of CPUID Extended Topology Leaf 10-37
10.9.3 Hierarchical ID of Logical Processors in an MP System 10-40
10.9.3.1 Hierarchical ID of Logical Processors with x2APIC ID 10-41
10.9.4 Algorithm for Three-Domain Mappings of APIC_ID 10-42
10.9.5 Identifying Topological Relationships in an MP System 10-46
10.10 MANAGEMENT OF IDLE AND BLOCKED CONDITIONS 10-49
10.10.1 HLT Instruction 10-49
10.10.2 PAUSE Instruction 10-49
10.10.3 Detecting Support MONITOR/MWAIT Instruction 10-50
10.10.4 MONITOR/MWAIT Instruction 10-50
10.10.5 Monitor/Mwait Address Range Determination 10-51
10.10.6 Required Operating System Support 10-52
10.10.6.1 Use the PAUSE Instruction in Spin-Wait Loops 10-52
10.10.6.2 Potential Usage of MONITOR/MWAIT in C0 Idle Loops 10-52
10.10.6.3 Halt Idle Logical Processors 10-54
10.10.6.4 Potential Usage of MONITOR/MWAIT in C1 Idle Loops 10-54
10.10.6.5 Guidelines for Scheduling Threads on Logical Processors Sharing Execution Resources 10-55
10.10.6.6 Eliminate Execution-Based Timing Loops 10-55
10.10.6.7 Place Locks and Semaphores in Aligned, 128-Byte Blocks of Memory 10-55
10.11 MP INITIALIZATION FOR P6 FAMILY PROCESSORS 10-55

10.11.1 Overview of the MP Initialization Process for P6 Family Processors 10-56
10.11.2 MP Initialization Protocol Algorithm 10-56
10.11.2.1 Error Detection and Handling During the MP Initialization Protocol 10-58

CHAPTER 11
PROCESSOR MANAGEMENT AND INITIALIZATION

11.1 INITIALIZATION OVERVIEW 11-1
11.1.1 Processor State After Reset 11-2
11.1.2 Processor Built-In Self-Test (BIST) 11-5
11.1.3 Model and Stepping Information 11-5
11.1.4 First Instruction Executed 11-5
11.2 X87 FPU INITIALIZATION 11-5
11.2.1 Configuring the x87 FPU Environment 11-6
11.2.2 Setting the Processor for x87 FPU Software Emulation 11-6
11.3 CACHE ENABLING 11-7
11.4 MODEL-SPECIFIC REGISTERS (MSRS) 11-7
11.5 MEMORY TYPE RANGE REGISTERS (MTRRS) 11-8
11.6 INITIALIZING SSE/SSE2/SSE3/SSSE3 EXTENSIONS 11-8
11.7 SOFTWARE INITIALIZATION FOR REAL-ADDRESS MODE OPERATION 11-8
11.7.1 Real-Address Mode IDT 11-8
11.7.2 NMI Interrupt Handling 11-9
11.8 SOFTWARE INITIALIZATION FOR PROTECTED-MODE OPERATION 11-9
11.8.1 Protected-Mode System Data Structures 11-9
11.8.2 Initializing Protected-Mode Exceptions and Interrupts 11-10
11.8.3 Initializing Paging 11-10
11.8.4 Initializing Multitasking 11-10
11.8.5 Initializing IA-32e Mode 11-11
11.8.5.1 IA-32e Mode System Data Structures 11-11
11.8.5.2 IA-32e Mode Interrupts and Exceptions 11-12
11.8.5.3 64-bit Mode and Compatibility Mode Operation 11-12
11.8.5.4 Switching Out of IA-32e Mode Operation 11-12
11.9 MODE SWITCHING 11-13
11.9.1 Switching to Protected Mode 11-13
11.9.2 Switching Back to Real-Address Mode 11-14
11.10 INITIALIZATION AND MODE SWITCHING EXAMPLE 11-14
11.10.1 Assembler Usage 11-16
11.10.2 STARTUP.ASM Listing 11-16
11.10.3 MAIN.ASM Source Code 11-25
11.10.4 Supporting Files 11-25
11.11 MICROCODE UPDATE FACILITIES 11-27
11.11.1 Microcode Update 11-28
11.11.2 Optional Extended Signature Table 11-31
11.11.3 Processor Identification 11-32
11.11.4 Platform Identification 11-32
11.11.5 Microcode Update Checksum 11-33
11.11.6 Microcode Update Loader 11-34
11.11.6.1 Hard Resets in Update Loading 11-35
11.11.6.2 Update in a Multiprocessor System 11-35
11.11.6.3 Update in a System Supporting Intel Hyper-Threading Technology 11-35
11.11.6.4 Update in a System Supporting Dual-Core Technology 11-35
11.11.6.5 Update Loader Enhancements 11-35
11.11.7 Update Signature and Verification 11-36
11.11.7.1 Determining the Signature 11-36
11.11.7.2 Authenticating the Update 11-37
11.11.8 Optional Processor Microcode Update Specifications 11-37
11.11.8.1 Responsibilities of the BIOS 11-38
11.11.8.2 Responsibilities of the Calling Program 11-39
11.11.8.3 Microcode Update Functions 11-42
11.11.8.4 INT 15H-based Interface 11-42
11.11.8.5 Function 00H—Presence Test 11-42
11.11.8.6 Function 01H—Write Microcode Update Data 11-43
11.11.8.7 Function 02H—Microcode Update Control 11-46
11.11.8.8 Function 03H—Read Microcode Update Data 11-47
11.11.8.9 Return Codes 11-48

PAGE

CHAPTER 12
ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC)

12.1 LOCAL AND I/O APIC OVERVIEW 12-1
12.2 SYSTEM BUS VS. APIC BUS 12-3
12.3 THE INTEL® 82489DX EXTERNAL APIC, THE APIC, THE XAPIC, AND THE X2APIC 12-4
12.4 LOCAL APIC 12-4
12.4.1 The Local APIC Block Diagram 12-4
12.4.2 Presence of the Local APIC 12-7
12.4.3 Enabling or Disabling the Local APIC 12-7
12.4.4 Local APIC Status and Location 12-8
12.4.5 Relocating the Local APIC Registers 12-9
12.4.6 Local APIC ID 12-9
12.4.7 Local APIC State 12-10
12.4.7.1 Local APIC State After Power-Up or Reset 12-10
12.4.7.2 Local APIC State After It Has Been Software Disabled 12-10
12.4.7.3 Local APIC State After an INIT Reset ("Wait-for-SIPI" State) 12-10
12.4.7.4 Local APIC State After It Receives an INIT-Deassert IPI 12-11
12.4.8 Local APIC Version Register 12-11
12.5 HANDLING LOCAL INTERRUPTS 12-12
12.5.1 Local Vector Table 12-12
12.5.2 Valid Interrupt Vectors 12-15
12.5.3 Error Handling 12-15
12.5.4 APIC Timer 12-16
12.5.4.1 TSC-Deadline Mode 12-17
12.5.5 Local Interrupt Acceptance 12-18
12.6 ISSUING INTERPROCESSOR INTERRUPTS 12-19
12.6.1 Interrupt Command Register (ICR) 12-19
12.6.2 Determining IPI Destination 12-22
12.6.2.1 Physical Destination Mode 12-23
12.6.2.2 Logical Destination Mode 12-23
12.6.2.3 Broadcast/Self Delivery Mode 12-25
12.6.2.4 Lowest Priority Delivery Mode 12-25
12.6.3 IPI Delivery and Acceptance 12-26
12.7 SYSTEM AND APIC BUS ARBITRATION 12-26
12.8 HANDLING INTERRUPTS 12-26
12.8.1 Interrupt Handling with the Pentium 4 and Intel Xeon Processors 12-27
12.8.2 Interrupt Handling with the P6 Family and Pentium Processors 12-27
12.8.3 Interrupt, Task, and Processor Priority 12-28
12.8.3.1 Task and Processor Priorities 12-29
12.8.4 Interrupt Acceptance for Fixed Interrupts 12-30
12.8.5 Signaling Interrupt Servicing Completion 12-31
12.8.6 Task Priority in IA-32e Mode 12-31
12.8.6.1 Interaction of Task Priorities between CR8 and APIC 12-32
12.9 SPURIOUS INTERRUPT 12-32
12.10 APIC BUS MESSAGE PASSING MECHANISM AND PROTOCOL (P6 FAMILY, PENTIUM PROCESSORS) 12-33
12.10.1 Bus Message Formats 12-34
12.11 MESSAGE SIGNALLED INTERRUPTS 12-34
12.11.1 Message Address Register Format 12-34
12.11.2 Message Data Register Format 12-35
12.12 EXTENDED XAPIC (X2APIC) 12-36
12.12.1 Detecting and Enabling x2APIC Mode 12-37
12.12.1.1 Instructions to Access APIC Registers 12-37
12.12.1.2 x2APIC Register Address Space 12-38
12.12.1.3 Reserved Bit Checking 12-40
12.12.2 x2APIC Register Availability 12-40
12.12.3 MSR Access in x2APIC Mode 12-40
12.12.4 VM-Exit Controls for MSRs and x2APIC Registers 12-41
12.12.5 x2APIC State Transitions 12-41
12.12.5.1 x2APIC States 12-41
x2APIC After Reset 12-42
x2APIC Transitions From x2APIC Mode 12-42
x2APIC Transitions From Disabled Mode 12-43
State Changes From xAPIC Mode to x2APIC Mode 12-43
12.12.6 Routing of Device Interrupts in x2APIC Mode 12-43

12.12.7 Initialization by System Software 12-43
12.12.8 CPUID Extensions And Topology Enumeration....12-43
12.12.8.1 Consistency of APIC IDs and CPUID 12-44
12.12.9 ICR Operation in x2APIC Mode....12-44
12.12.10 Determining IPI Destination in x2APIC Mode 12-45
12.12.10.1 Logical Destination Mode in x2APIC Mode....12-45
12.12.10.2 Deriving Logical x2APIC ID from the Local x2APIC ID....12-46
12.12.11 SELF IPI Register 12-47
12.13 APIC BUS MESSAGE FORMATS....12-47
12.13.1 Bus Message Formats 12-47
12.13.2 EOI Message 12-47
12.13.2.1 Short Message 12-48
12.13.2.2 Non-focused Lowest Priority Message....12-49
12.13.2.3 APIC Bus Status Cycles 12-50

CHAPTER 13
MEMORY CACHE CONTROL

13.1 INTERNAL CACHES, TLBS, AND BUFFERS 13-1
13.2 CACHING TERMINOLOGY 13-5
13.3 METHODS OF CACHING AVAILABLE....13-6
13.3.1 Buffering of Write Combining Memory Locations 13-8
13.3.2 Choosing a Memory Type 13-8
13.3.3 Code Fetches in Uncacheable Memory 13-9
13.4 CACHE CONTROL PROTOCOL....13-9
13.5 CACHE CONTROL 13-10
13.5.1 Cache Control Registers and Bits 13-10
13.5.2 Precedence of Cache Controls....13-13
13.5.2.1 Selecting Memory Types for Pentium Pro and Pentium II Processors 13-14
13.5.2.2 Selecting Memory Types for Pentium III and More Recent Processor Families 13-15
13.5.2.3 Writing Values Across Pages with Different Memory Types....13-16
13.5.3 Preventing Caching....13-16
13.5.4 Disabling and Enabling the L3 Cache 13-17
13.5.5 Cache Management Instructions....13-17
13.5.6 L1 Data Cache Context Mode....13-18
13.5.6.1 Adaptive Mode 13-18
13.5.6.2 Shared Mode....13-18
13.6 SELF-MODIFYING CODE 13-18
13.7 IMPLICIT CACHING (PENTIUM 4, INTEL® XEON®, AND P6 FAMILY PROCESSORS)....13-19
13.8 EXPLICIT CACHING....13-19
13.9 INVALIDATING THE TRANSLATION LOOKASIDE BUFFERS (TLBS)....13-19
13.10 STORE BUFFER....13-20
13.11 MEMORY TYPE RANGE REGISTERS (MTRRS)....13-20
13.11.1 MTRR Feature Identification 13-21
13.11.2 Setting Memory Ranges with MTRRs 13-22
13.11.2.1 IA32_MTRR_DEF_TYPE MSR....13-22
13.11.2.2 Fixed Range MTRRs 13-23
13.11.2.3 Variable Range MTRRs....13-23
13.11.2.4 System-Management Range Register Interface 13-25
13.11.3 Example Base and Mask Calculations 13-26
13.11.3.1 Base and Mask Calculations for Greater-Than 36-bit Physical Address Support....13-27
13.11.4 Range Size and Alignment Requirement 13-28
13.11.4.1 MTRR Precedences....13-28
13.11.5 MTRR Initialization....13-29
13.11.6 Remapping Memory Types....13-29
13.11.7 MTRR Maintenance Programming Interface 13-29
13.11.7.1 MemTypeGet() Function 13-29
13.11.7.2 MemTypeSet() Function 13-31
13.11.8 MTRR Considerations in MP Systems 13-32
13.11.9 Large Page Size Considerations 13-33
13.12 PAGE ATTRIBUTE TABLE (PAT)....13-33
13.12.1 Detecting Support for the PAT Feature....13-34
13.12.2 IA32_PAT MSR....13-34
13.12.3 Selecting a Memory Type from the PAT 13-35
13.12.4 Programming the PAT 13-35

PAGE

13.12.5 PAT Compatibility with Earlier IA-32 Processors 13-36

CHAPTER 14
INTEL® MMX™ TECHNOLOGY SYSTEM PROGRAMMING

14.1 EMULATION OF THE MMX INSTRUCTION SET 14-1
14.2 THE MMX STATE AND MMX REGISTER ALIASING 14-1
14.2.1 Effect of MMX, x87 FPU, FXSAVE, and FXRSTOR Instructions on the x87 FPU Tag Word 14-3
14.3 SAVING AND RESTORING THE MMX STATE AND REGISTERS 14-3
14.4 SAVING MMX STATE ON TASK OR CONTEXT SWITCHES 14-4
14.5 EXCEPTIONS THAT CAN OCCUR WHEN EXECUTING MMX INSTRUCTIONS 14-4
14.5.1 Effect of MMX Instructions on Pending x87 Floating-Point Exceptions 14-5
14.6 DEBUGGING MMX CODE 14-5

CHAPTER 15
SYSTEM PROGRAMMING FOR INSTRUCTION SET EXTENSIONS AND PROCESSOR EXTENDED STATES

15.1 PROVIDING OPERATING SYSTEM SUPPORT FOR SSE EXTENSIONS 15-1
15.1.1 Adding Support to an Operating System for SSE Extensions 15-2
15.1.2 Checking for CPU Support 15-2
15.1.3 Initialization of the SSE Extensions 15-2
15.1.4 Providing Non-Numeric Exception Handlers for Exceptions Generated by the SSE Instructions 15-4
15.1.5 Providing a Handler for the SIMD Floating-Point Exception (#XM) 15-5
15.1.5.1 Numeric Error flag and IGNNE# 15-6
15.2 EMULATION OF SSE EXTENSIONS 15-6
15.3 SAVING AND RESTORING SSE STATE 15-6
15.4 DESIGNING OS FACILITIES FOR SAVING X87 FPU, SSE, AND EXTENDED STATES ON TASK OR CONTEXT SWITCHES 15-6
15.4.1 Using the TS Flag to Control the Saving of the x87 FPU and SSE State 15-7
15.5 THE XSAVE FEATURE SET AND PROCESSOR EXTENDED STATE MANAGEMENT 15-7
15.5.1 Checking the Support for XSAVE Feature Set 15-8
15.5.2 Determining the XSAVE Managed Feature States And The Required Buffer Size 15-8
15.5.3 Enable the Use Of XSAVE Feature Set And XSAVE State Components 15-9
15.5.4 Provide an Initialization for the XSAVE State Components 15-9
15.5.5 Providing the Required Exception Handlers 15-9
15.6 INTEROPERABILITY OF THE XSAVE FEATURE SET AND FXSAVE/FXRSTOR 15-9
15.7 THE XSAVE FEATURE SET AND PROCESSOR SUPERVISOR STATE MANAGEMENT 15-10
15.8 SYSTEM PROGRAMMING FOR XSAVE MANAGED FEATURES 15-10
15.8.1 Intel® Advanced Vector Extensions (Intel® AVX) 15-11
15.8.2 Intel® Advanced Vector Extensions 512 (Intel® AVX-512) 15-11

CHAPTER 16
POWER AND THERMAL MANAGEMENT

16.1 ENHANCED INTEL SPEEDSTEP® TECHNOLOGY 16-1
16.1.1 Software Interface For Initiating Performance State Transitions 16-1
16.2 P-STATE HARDWARE COORDINATION 16-1
16.3 SYSTEM SOFTWARE CONSIDERATIONS AND OPPORTUNISTIC PROCESSOR PERFORMANCE OPERATION 16-3
16.3.1 Intel® Dynamic Acceleration Technology 16-3
16.3.2 System Software Interfaces for Opportunistic Processor Performance Operation 16-3
16.3.2.1 Discover Hardware Support and Enabling of Opportunistic Processor Performance Operation 16-3
16.3.2.2 OS Control of Opportunistic Processor Performance Operation 16-4
16.3.2.3 Required Changes to OS Power Management P-State Policy 16-4
16.3.3 Intel® Turbo Boost Technology 16-5
16.3.4 Performance and Energy Bias Hint Support 16-5
16.4 HARDWARE-CONTROLLED PERFORMANCE STATES (HWP) 16-5
16.4.1 HWP Programming Interfaces 16-6
16.4.2 Enabling HWP 16-7
16.4.3 HWP Performance Range and Dynamic Capabilities 16-7
16.4.4 Managing HWP 16-8
16.4.4.1 IA32_HWP_REQUEST MSR (Address: 774H Logical Processor Scope) 16-8
16.4.4.2 IA32_HWP_REQUEST_PKG MSR (Address: 772H Package Scope) 16-11
16.4.4.3 IA32_HWP_PECI_REQUEST_INFO MSR (Address 775H Package Scope) 16-11
16.4.4.4 IA32_HWP_CTL MSR (Address: 776H Logical Processor Scope) 16-12

PAGE

16.4.5 HWP Feedback 16-13
16.4.5.1 Non-Architectural HWP Feedback 16-15
16.4.6 HWP Notifications 16-16
16.4.7 Idle Logical Processor Impact on Core Frequency 16-16
16.4.8 Fast Write of Uncore MSR (Model Specific Feature) 16-17
16.4.8.1 FAST_UNCORE_MSRS_CAPABILITY (Address: 65FH, Logical Processor Scope) 16-17
16.4.8.2 FAST_UNCORE_MSRS_CTL (Address: 657H, Logical Processor Scope) 16-17
16.4.8.3 FAST_UNCORE_MSRS_STATUS (Address: 65EH, Logical Processor Scope) 16-18
16.4.9 Fast_IA32_HWP_REQUEST CPUID 16-18
16.4.10 Recommendations for OS use of HWP Controls 16-18
16.5 HARDWARE DUTY CYCLING (HDC) 16-20
16.5.1 Hardware Duty Cycling Programming Interfaces 16-20
16.5.2 Package level Enabling HDC 16-21
16.5.3 Logical-Processor Level HDC Control 16-22
16.5.4 HDC Residency Counters 16-22
16.5.4.1 IA32_THREAD_STALL 16-22
16.5.4.2 Non-Architectural HDC Residency Counters 16-23
16.5.5 MPERF and APERF Counters Under HDC 16-25
16.6 HARDWARE FEEDBACK INTERFACE AND INTEL® THREAD DIRECTOR 16-25
16.6.1 Hardware Feedback Interface Table Structure 16-25
16.6.2 Intel® Thread Director Table Structure 16-27
16.6.3 Intel® Thread Director Usage Model 16-30
16.6.4 Hardware Feedback Interface Pointer 16-31
16.6.5 Hardware Feedback Interface Configuration 16-31
16.6.6 Hardware Feedback Interface Notifications 16-32
16.6.7 Hardware Feedback Interface and Intel® Thread Director Structure Dynamic Update 16-33
16.6.8 Logical Processor Scope Intel® Thread Director Configuration 16-33
16.6.9 Implicit Reset of Package and Logical Processor Scope Configuration MSRs 16-34
16.6.10 Logical Processor Scope Intel® Thread Director Run Time Characteristics 16-34
16.6.11 Logical Processor Scope History 16-34
16.6.11.1 Enabling Intel® Thread Director History Reset 16-35
16.6.11.2 Implicit Intel® Thread Director History Reset 16-35
16.7 MWAIT EXTENSIONS FOR ADVANCED POWER MANAGEMENT 16-35
16.8 THERMAL MONITORING AND PROTECTION 16-36
16.8.1 Catastrophic Shutdown Detector 16-37
16.8.2 Thermal Monitor 16-37
16.8.2.1 Thermal Monitor 1 16-37
16.8.2.2 Thermal Monitor 2 16-37
16.8.2.3 Two Methods for Enabling TM2 16-37
16.8.2.4 Performance State Transitions and Thermal Monitoring 16-38
16.8.2.5 Thermal Status Information 16-38
16.8.2.6 Adaptive Thermal Monitor 16-39
16.8.3 Software Controlled Clock Modulation 16-40
16.8.3.1 Extension of Software Controlled Clock Modulation 16-41
16.8.4 Detection of Thermal Monitor and Software Controlled Clock Modulation Facilities 16-41
16.8.4.1 Detection of Software Controlled Clock Modulation Extension 16-41
16.8.5 On Die Digital Thermal Sensors 16-42
16.8.5.1 Digital Thermal Sensor Enumeration 16-42
16.8.5.2 Reading the Digital Sensor 16-42
16.8.6 Power Limit Notification 16-45
16.9 PACKAGE LEVEL THERMAL MANAGEMENT 16-45
16.9.1 Support for Passive and Active cooling 16-47
16.10 PLATFORM SPECIFIC POWER MANAGEMENT SUPPORT 16-48
16.10.1 RAPL Interfaces 16-48
16.10.2 RAPL Domains and Platform Specificity 16-49
16.10.3 Package RAPL Domain 16-50
16.10.4 PP0/PP1 RAPL Domains 16-52
16.10.5 DRAM RAPL Domain 16-54

CHAPTER 17
MACHINE-CHECK ARCHITECTURE

17.1 MACHINE-CHECK ARCHITECTURE 17-1
17.2 COMPATIBILITY WITH PENTIUM PROCESSOR 17-1
17.3 MACHINE-CHECK MSRS 17-2

PAGE

17.3.1 Machine-Check Global Control MSRs....17-2
17.3.1.1 IA32_MCG_CAP MSR....17-2
17.3.1.2 IA32_MCG_STATUS MSR....17-4
17.3.1.3 IA32_MCG_CTL MSR....17-4
17.3.1.4 IA32_MCG_EXT_CTL MSR....17-4
17.3.1.5 Enabling Local Machine Check....17-5
17.3.2 Error-Reporting Register Banks....17-5
17.3.2.1 IA32_MCi_CTL MSRs....17-5
17.3.2.2 IA32_MCi_STATUS MSRS....17-6
17.3.2.3 IA32_MCi_ADDR MSRs....17-9
17.3.2.4 IA32_MCi_MISC MSRs....17-9
17.3.2.5 IA32_MCi_CTL2 MSRs....17-11
17.3.2.6 IA32_MCG Extended Machine Check State MSRs....17-12
17.3.3 Mapping of the Pentium Processor Machine-Check Errors to the Machine-Check Architecture....17-13
17.4 ENHANCED CACHE ERROR REPORTING....17-13
17.5 CORRECTED MACHINE CHECK ERROR INTERRUPT....17-14
17.5.1 CMCI Local APIC Interface....17-14
17.5.2 System Software Recommendation for Managing CMCI and Machine Check Resources....17-15
17.5.2.1 CMCI Initialization....17-15
17.5.2.2 CMCI Threshold Management....17-16
17.5.2.3 CMCI Interrupt Handler....17-16
17.6 RECOVERY OF UNCORRECTED RECOVERABLE (UCR) ERRORS....17-16
17.6.1 Detection of Software Error Recovery Support....17-16
17.6.2 UCR Error Reporting and Logging....17-16
17.6.3 UCR Error Classification....17-17
17.6.4 UCR Error Overwrite Rules....17-18
17.7 MACHINE-CHECK AVAILABILITY....17-19
17.8 MACHINE-CHECK INITIALIZATION....17-19
17.9 INTERPRETING THE MCA ERROR CODES....17-20
17.9.1 Simple Error Codes....17-20
17.9.2 Compound Error Codes....17-21
17.9.2.1 Correction Report Filtering (F) Bit....17-21
17.9.2.2 Transaction Type (TT) Sub-Field....17-22
17.9.2.3 Level (LL) Sub-Field....17-22
17.9.2.4 Request (RRRR) Sub-Field....17-22
17.9.2.5 Bus and Interconnect Errors....17-23
17.9.2.6 Memory Controller and Extended Memory Errors....17-23
17.9.3 Architecturally Defined UCR Errors....17-24
17.9.3.1 Architecturally Defined SRAO Errors....17-24
17.9.3.2 Architecturally Defined SRAR Errors....17-25
17.9.4 Multiple MCA Errors....17-27
17.9.5 Machine-Check Error Codes Interpretation....17-28
17.10 GUIDELINES FOR WRITING MACHINE-CHECK SOFTWARE....17-28
17.10.1 Machine-Check Exception Handler....17-28
17.10.2 Pentium Processor Machine-Check Exception Handling....17-29
17.10.3 Logging Correctable Machine-Check Errors....17-30
17.10.4 Machine-Check Software Handler Guidelines for Error Recovery....17-31
17.10.4.1 Machine-Check Exception Handler for Error Recovery....17-31
17.10.4.2 Corrected Machine-Check Handler for Error Recovery....17-35

CHAPTER 18
INTERPRETING MACHINE CHECK ERROR CODES

18.1 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 06H, MACHINE ERROR CODES FOR MACHINE CHECK... 18-1
18.2 INCREMENTAL DECODING INFORMATION: INTEL® CORE™ 2 PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK....18-3
18.2.1 Model-Specific Machine Check Error Codes for Intel® Xeon® Processor 7400 Series....18-5
18.2.1.1 Processor Machine Check Status Register, Incremental MCA Error Code Definition....18-6
18.2.2 Intel® Xeon® Processor 7400 Model Specific Error Code Field....18-6
18.2.2.1 Processor Model Specific Error Code Field, Type B: Bus and Interconnect Error Codes....18-6
18.2.2.2 Processor Model Specific Error Code Field, Type C: Cache Bus Controller Error Codes....18-7
18.3 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR 3400, 3500, 5500 SERIES, MACHINE ERROR CODES FOR MACHINE CHECK....18-7
18.3.1 Intel® QPI Machine Check Errors....18-8
18.3.2 Internal Machine Check Errors....18-9

PAGE

18.3.3 Memory Controller Errors . . . 18-9
18.4 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK . . . 18-10
18.4.1 Internal Machine Check Errors . . . 18-10
18.4.2 Intel® QPI Machine Check Errors . . . 18-11
18.4.3 Integrated Memory Controller Machine Check Errors . . . 18-11
18.5 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V2 AND INTEL® XEON® PROCESSOR E7 V2 FAMILIES, MACHINE ERROR CODES FOR MACHINE CHECK . . . 18-13
18.5.1 Internal Machine Check Errors . . . 18-13
18.5.2 Integrated Memory Controller Machine Check Errors . . . 18-14
18.5.3 Home Agent Machine Check Errors . . . 18-15
18.6 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V3 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK . . . 18-15
18.6.1 Internal Machine Check Errors . . . 18-16
18.6.2 Intel® QPI Machine Check Errors . . . 18-17
18.6.3 Integrated Memory Controller Machine Check Errors . . . 18-17
18.6.4 Home Agent Machine Check Errors . . . 18-19
18.7 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR D FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK . . . 18-19
18.7.1 Internal Machine Check Errors . . . 18-19
18.7.2 Integrated Memory Controller Machine Check Errors . . . 18-20
18.8 INCREMENTAL DECODING INFORMATION: INTEL® XEON® PROCESSOR E5 V4 FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK . . . 18-21
18.8.1 Integrated Memory Controller Machine Check Errors . . . 18-21
18.8.2 Home Agent Machine Check Errors . . . 18-21
18.9 INCREMENTAL DECODING INFORMATION: INTEL® XEON® SCALABLE PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK . . . 18-22
18.9.1 Internal Machine Check Errors . . . 18-22
18.9.2 Interconnect Machine Check Errors . . . 18-24
18.9.3 Integrated Memory Controller Machine Check Errors . . . 18-25
18.9.4 M2M Machine Check Errors . . . 18-26
18.9.5 Home Agent Machine Check Errors . . . 18-27
18.10 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUID DISPLAYFAMILY_DISPLAYMODEL SIGNATURE 06_5FH, MACHINE ERROR CODES FOR MACHINE CHECK . . . 18-28
18.10.1 Integrated Memory Controller Machine Check Errors . . . 18-28
18.11 INCREMENTAL DECODING INFORMATION: 3RD GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK . . . 18-28
18.11.1 Internal Machine Check Errors . . . 18-29
18.11.2 Interconnect Machine Check Errors . . . 18-31
18.11.3 Integrated Memory Controller Machine Check Errors . . . 18-32
18.11.4 M2M Machine Check Errors . . . 18-36
18.12 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUID DISPLAYFAMILY_DISPLAYMODEL SIGNATURE 06_86H, MACHINE ERROR CODES FOR MACHINE CHECK . . . 18-36
18.12.1 Integrated Memory Controller Machine Check Errors . . . 18-36
18.12.2 M2M Machine Check Errors . . . 18-37
18.13 INCREMENTAL DECODING INFORMATION: 4TH GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY, MACHINE ERROR CODES FOR MACHINE CHECK . . . 18-37
18.13.1 Internal Machine Check Errors . . . 18-37
18.13.2 Interconnect Machine Check Errors . . . 18-39
18.13.3 Integrated Memory Controller Machine Check Errors . . . 18-41
18.13.4 M2M Machine Check Errors . . . 18-43
18.13.5 High Bandwidth Memory Machine Check Errors . . . 18-44
18.14 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 0FH, MACHINE ERROR CODES FOR MACHINE CHECK . . . 18-44
18.14.1 Model-Specific Machine Check Error Codes for the Intel® Xeon® Processor MP 7100 Series . . . 18-45
18.14.1.1 Processor Machine Check Status Register MCA Error Code Definition . . . 18-46
18.14.2 Other_Info Field (All MCA Error Types) . . . 18-47
18.14.3 Processor Model Specific Error Code Field . . . 18-48
18.14.3.1 MCA Error Type A: L3 Error . . . 18-48
18.14.3.2 Processor Model Specific Error Code Field Type B: Bus and Interconnect Error . . . 18-48
18.14.3.3 Processor Model Specific Error Code Field Type C: Cache Bus Controller Error . . . 18-49

PAGE

CHAPTER 19
DEBUG, BRANCH PROFILE, TSC, AND INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) FEATURES

19.1 OVERVIEW OF DEBUG SUPPORT FACILITIES 19-1
19.2 DEBUG REGISTERS 19-2
19.2.1 Debug Address Registers (DR0-DR3) 19-4
19.2.2 Debug Registers DR4 and DR5 19-4
19.2.3 Debug Status Register (DR6) 19-4
19.2.4 Debug Control Register (DR7) 19-4
19.2.5 Breakpoint Field Recognition 19-6
19.2.6 Debug Registers and Intel® 64 Processors 19-7
19.3 DEBUG EXCEPTIONS 19-7
19.3.1 Debug Exception (#DB)—Interrupt Vector 1 19-7
19.3.1.1 Instruction-Breakpoint Exception Condition 19-9
19.3.1.2 Data Memory and I/O Breakpoint Exception Conditions 19-10
19.3.1.3 General-Detect Exception Condition 19-10
19.3.1.4 Single-Step Exception Condition 19-11
19.3.1.5 Task-Switch Exception Condition 19-11
19.3.1.6 OS Bus-Lock Detection 19-11
19.3.2 Breakpoint Exception (#BP)—Interrupt Vector 3 19-11
19.3.3 Debug Exceptions, Breakpoint Exceptions, and Restricted Transactional Memory (RTM) 19-12
19.4 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING OVERVIEW 19-12
19.4.1 IA32_DEBUGCTL MSR 19-13
19.4.2 Monitoring Branches, Exceptions, and Interrupts 19-14
19.4.3 Single-Stepping on Branches 19-14
19.4.4 Branch Trace Messages 19-15
19.4.4.1 Branch Trace Message Visibility 19-15
19.4.5 Branch Trace Store (BTS) 19-15
19.4.6 CPL-Qualified Branch Trace Mechanism 19-15
19.4.7 Freezing LBR and Performance Counters on PMI 19-15
19.4.8 LBR Stack 19-17
19.4.8.1 LBR Stack and Intel® 64 Processors 19-18
19.4.8.2 LBR Stack and IA-32 Processors 19-19
19.4.8.3 Last Exception Records and Intel 64 Architecture 19-19
19.4.9 BTS and DS Save Area 19-19
19.4.9.1 64 Bit Format of the DS Save Area 19-22
19.4.9.2 Setting Up the DS Save Area 19-24
19.4.9.3 Setting Up the BTS Buffer 19-25
19.4.9.4 Setting Up CPL-Qualified BTS 19-26
19.4.9.5 Writing the DS Interrupt Service Routine 19-26
19.5 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ 2 DUO AND INTEL ATOM® PROCESSORS) 19-27
19.5.1 LBR Stack 19-27
19.5.2 LBR Stack in Intel Atom® Processors based on the Silvermont Microarchitecture 19-28
19.6 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT MICROARCHITECTURE 19-28
19.7 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT PLUS MICROARCHITECTURE 19-29
19.8 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR INTEL® XEON PHI™ PROCESSOR 7200/5200/3200 19-29
19.9 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON NEHALEM MICROARCHITECTURE 19-29
19.9.1 LBR Stack 19-30
19.9.2 Filtering of Last Branch Records 19-31
19.10 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON SANDY BRIDGE MICROARCHITECTURE 19-31
19.11 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON HASWELL MICROARCHITECTURE 19-32
19.11.1 LBR Stack Enhancement 19-33
19.12 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON SKYLAKE MICROARCHITECTURE 19-33
19.12.1 MSR_LBR_INFO_x MSR 19-34
19.12.2 Streamlined Freeze_LBRs_On_PMI Operation 19-34
19.12.3 LBR Behavior and Deep C-State 19-35
19.13 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PROCESSORS BASED ON INTEL NETBURST® MICROARCHITECTURE) 19-35

19.13.1 MSR_DEBUGCTLA MSR 19-35
19.13.2 LBR Stack for Processors Based on Intel NetBurst® Microarchitecture 19-36
19.13.3 Last Exception Records 19-37
19.14 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS) 19-38
19.15 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PENTIUM M PROCESSORS) 19-39
19.16 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (P6 FAMILY PROCESSORS) 19-40
19.16.1 DEBUGCTLMSR Register 19-40
19.16.2 Last Branch and Last Exception MSRs 19-41
19.16.3 Monitoring Branches, Exceptions, and Interrupts 19-42
19.17 TIME-STAMP COUNTER 19-42
19.17.1 Invariant TSC 19-43
19.17.2 IA32_TSC_AUX Register and RDTSCP Support 19-43
19.17.3 Time-Stamp Counter Adjustment 19-44
19.17.4 Invariant Time-Keeping 19-44
19.18 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) MONITORING FEATURES 19-44
19.18.1 Overview of Cache Monitoring Technology and Memory Bandwidth Monitoring 19-45
19.18.2 Enabling Monitoring: Usage Flow 19-45
19.18.3 Enumeration and Detecting Support of Cache Monitoring Technology and Memory Bandwidth Monitoring 19-46
19.18.4 Monitoring Resource Type and Capability Enumeration 19-46
19.18.5 Feature-Specific Enumeration 19-47
19.18.5.1 Cache Monitoring Technology 19-48
19.18.5.2 Memory Bandwidth Monitoring 19-48
19.18.6 Monitoring Resource RMID Association 19-49
19.18.7 Monitoring Resource Selection and Reporting Infrastructure 19-50
19.18.8 Monitoring Programming Considerations 19-51
19.18.8.1 Monitoring Dynamic Configuration 19-51
19.18.8.2 Monitoring Operation With Power Saving Features 19-52
19.18.8.3 Monitoring Operation with Other Operating Modes 19-52
19.18.8.4 Monitoring Operation with RAS Features 19-52
19.19 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) ALLOCATION FEATURES 19-52
19.19.1 Introduction to Cache Allocation Technology (CAT) 19-52
19.19.2 Cache Allocation Technology Architecture 19-53
19.19.3 Code and Data Prioritization (CDP) Technology 19-56
19.19.4 Enabling Cache Allocation Technology Usage Flow 19-57
19.19.4.1 Enumeration and Detection Support of Cache Allocation Technology 19-58
19.19.4.2 Cache Allocation Technology: Resource Type and Capability Enumeration 19-58
19.19.4.3 Cache Allocation Technology: Cache Mask Configuration 19-61
19.19.4.4 Class of Service to Cache Mask Association: Common Across Allocation Features 19-62
19.19.5 Code and Data Prioritization (CDP): Enumerating and Enabling L3 CDP Technology 19-62
19.19.5.1 Mapping Between L3 CDP Masks and CAT Masks 19-63
19.19.6 Code and Data Prioritization (CDP): Enumerating and Enabling L2 CDP Technology 19-63
19.19.6.1 Mapping Between L2 CDP Masks and L2 CAT Masks 19-64
19.19.6.2 Common L2 and L3 CDP Programming Considerations 19-64
19.19.6.3 Cache Allocation Technology Dynamic Configuration 19-64
19.19.6.4 Cache Allocation Technology Operation With Power Saving Features 19-65
19.19.6.5 Cache Allocation Technology Operation with Other Operating Modes 19-65
19.19.6.6 Associating Threads with CAT/CDP Classes of Service 19-65
19.19.7 Introduction to Memory Bandwidth Allocation 19-66
19.19.7.1 Memory Bandwidth Allocation Enumeration 19-66
19.19.7.2 Memory Bandwidth Allocation Configuration 19-67
19.19.7.3 Memory Bandwidth Allocation Usage Considerations 19-68
19.20 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) FOR NON-CPU AGENTS 19-69
19.20.1 Non-CPU Agent Intel® RDT Features Enumeration Details 19-69
19.20.1.1 CPUID-Based Enumeration for Non-CPU Agent Intel® RDT Feature 19-69
19.20.1.2 ACPI Enumeration 19-70
19.20.2 Non-CPU Agent Intel® RDT Feature Enable MSR 19-70

CHAPTER 20
LAST BRANCH RECORDS

20.1 BEHAVIOR 20-1
20.1.1 Logged Operations 20-1
20.1.2 Configuration 20-2
20.1.2.1 Enabling and Disabling 20-2

PAGE

20.1.2.2 LBR Depth 20-2
20.1.2.3 Branch Type Enabling and Filtering 20-2
20.1.2.4 Call-Stack Mode 20-3
Call-Stack Mode and LBR Freeze 20-3
20.1.2.5 CPL Filtering 20-4
20.1.3 Record Data 20-4
20.1.3.1 IP Fields 20-4
20.1.3.2 Branch Types 20-4
20.1.3.3 Cycle Time 20-4
20.1.3.4 Mispredict Information 20-5
20.1.3.5 Intel® TSX Information 20-5
20.1.3.6 LBR Event Logging 20-5
20.1.4 Interaction with Other Processor Features 20-5
20.1.4.1 SMM 20-5
20.1.4.2 SMM Transfer Monitor (STM) 20-5
20.1.4.3 VMX 20-6
20.1.4.4 Intel® SGX 20-6
20.1.4.5 Debug Exceptions 20-6
20.1.4.6 SMX 20-6
20.1.4.7 MWAIT 20-6
20.1.4.8 Processor Event-Based Sampling (PEBS) 20-6
20.2 MSRS 20-7
20.3 FAST LBR READ ACCESS 20-7
20.4 OTHER IMPACTS 20-7
20.4.1 Branch Trace Store on Intel Atom® Processors 20-7
20.4.2 IA32_DEBUGCTL 20-7
20.4.3 IA32_PERF_CAPABILITIES 20-7

CHAPTER 21
PERFORMANCE MONITORING

21.1 PERFORMANCE MONITORING OVERVIEW 21-1
21.2 ARCHITECTURAL PERFORMANCE MONITORING 21-2
21.2.1 Architectural Performance Monitoring Version 1 21-3
21.2.2 Architectural Performance Monitoring Version 2 21-5
21.2.3 Architectural Performance Monitoring Version 3 21-9
21.2.3.1 AnyThread Counting and Software Evolution 21-11
21.2.4 Architectural Performance Monitoring Version 4 21-12
21.2.4.1 Enhancement in IA32_PERF_GLOBAL_STATUS 21-12
21.2.4.2 IA32_PERF_GLOBAL_STATUS_RESET and IA32_PERF_GLOBAL_STATUS_SET MSRS 21-13
21.2.4.3 IA32_PERF_GLOBAL_INUSE MSR 21-14
21.2.5 Architectural Performance Monitoring Version 5 21-15
21.2.5.1 AnyThread Mode Deprecation 21-15
21.2.5.2 Fixed Counter Enumeration 21-15
21.2.5.3 Domain Separation 21-15
21.2.6 Architectural Performance Monitoring Version 6 21-15
21.2.6.1 Performance Monitoring MSR Aliasing 21-16
21.2.6.2 Unit Mask 2 21-18
21.2.6.3 Equal Flag 21-18
21.2.7 Pre-defined Architectural Performance Events 21-18
21.2.8 Full-Width Writes to Performance Counter Registers 21-20
21.2.9 Scalable Enumeration Architecture 21-21
21.2.9.1 CPUID Sub-Leafing 21-21
21.2.9.2 Reporting Per Logical Processor 21-21
21.2.9.3 General-Purpose Counters Bitmap 21-22
21.2.9.4 Fixed-Function Counters True-View Bitmap 21-22
21.2.9.5 Architectural Performance Monitoring Events Bitmap 21-22
21.2.9.6 TMA Slots Per Cycle 21-22
21.3 PERFORMANCE MONITORING (INTEL® CORE™ PROCESSORS AND INTEL® XEON® PROCESSORS) 21-23
21.3.1 Performance Monitoring for Processors Based on Nehalem Microarchitecture 21-23
21.3.1.1 Enhancements of Performance Monitoring in the Processor Core 21-24
21.3.1.2 Performance Monitoring Facility in the Uncore 21-31
21.3.1.3 Intel® Xeon® Processor 7500 Series Performance Monitoring Facility 21-36
21.3.2 Performance Monitoring for Processors Based on Westmere Microarchitecture 21-37

PAGE

21.3.3 Intel® Xeon® Processor E7 Family Performance Monitoring Facility 21-38
21.3.4 Performance Monitoring for Processors Based on Sandy Bridge Microarchitecture 21-38
21.3.4.1 Global Counter Control Facilities in Sandy Bridge Microarchitecture 21-39
21.3.4.2 Counter Coalescence 21-41
21.3.4.3 Full Width Writes to Performance Counters 21-41
21.3.4.4 PEBS Support in Sandy Bridge Microarchitecture 21-41
21.3.4.5 Off-core Response Performance Monitoring 21-46
21.3.4.6 Uncore Performance Monitoring Facilities in the Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, and Intel® Core™ i3-2xxx Processor Series 21-49
21.3.4.7 Intel® Xeon® Processor E5 Family Performance Monitoring Facility 21-51
21.3.4.8 Intel® Xeon® Processor E5 Family Uncore Performance Monitoring Facility 21-52
21.3.5 3rd Generation Intel® Core™ Processor Performance Monitoring Facility 21-52
21.3.5.1 Intel® Xeon® Processor E5 v2 and E7 v2 Family Uncore Performance Monitoring Facility 21-52
21.3.6 4th Generation Intel® Core™ Processor Performance Monitoring Facility 21-52
21.3.6.1 Processor Event Based Sampling (PEBS) Facility 21-53
21.3.6.2 PEBS Data Format 21-54
21.3.6.3 PEBS Data Address Profiling 21-55
21.3.6.4 Off-core Response Performance Monitoring 21-56
21.3.6.5 Performance Monitoring and Intel® TSX 21-58
21.3.6.6 Uncore Performance Monitoring Facilities in the 4th Generation Intel® Core™ Processors 21-60
21.3.6.7 Intel® Xeon® Processor E5 v3 Family Uncore Performance Monitoring Facility 21-60
21.3.7 5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility 21-61
21.3.8 6th Generation, 7th Generation and 8th Generation Intel® Core™ Processor Performance Monitoring Facility 21-62
21.3.8.1 Processor Event Based Sampling (PEBS) Facility 21-63
21.3.8.2 Frontend Retired Facility 21-66
21.3.8.3 Off-core Response Performance Monitoring 21-68
21.3.8.4 Uncore Performance Monitoring Facilities on Intel® Core™ Processors Based on Cannon Lake Microarchitecture 21-71
21.3.9 10th Generation Intel® Core™ Processor Performance Monitoring Facility 21-71
21.3.9.1 Processor Event Based Sampling (PEBS) Facility 21-72
21.3.9.2 Off-core Response Performance Monitoring 21-72
21.3.9.3 Performance Metrics 21-74
21.3.10 12th and 13th Generation Intel® Core™ Processors, and 4th and 5th Generation Intel® Xeon® Scalable Processor Family Performance Monitoring Facility 21-75
21.3.10.1 P-core Performance Monitoring Unit 21-75
21.3.10.2 E-core Performance Monitoring Unit 21-78
21.3.10.3 Unhalted Reference Cycles 21-80
21.3.11 Intel® Series 2 Core™ Ultra Processor Performance Monitoring Facility 21-81
21.3.11.1 P-core Performance Monitoring Unit 21-81
21.3.11.2 E-core Performance Monitoring Unit 21-82
21.4 PERFORMANCE MONITORING (INTEL® XEON™ PHI PROCESSORS) 21-82
21.4.1 Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring 21-83
21.4.1.1 Enhancements of Performance Monitoring in the Intel® Xeon Phi™ Processor Tile 21-83
21.5 PERFORMANCE MONITORING (INTEL ATOM® PROCESSORS) 21-87
21.5.1 Performance Monitoring (45 nm and 32 nm Intel Atom® Processors) 21-87
21.5.2 Performance Monitoring for Silvermont Microarchitecture 21-87
21.5.2.1 Enhancements of Performance Monitoring in the Processor Core 21-87
21.5.2.2 Offcore Response Event 21-89
21.5.2.3 Average Offcore Request Latency Measurement 21-92
21.5.3 Performance Monitoring for Goldmont Microarchitecture 21-92
21.5.3.1 Processor Event Based Sampling (PEBS) 21-93
21.5.3.2 Offcore Response Event 21-96
21.5.3.3 Average Offcore Request Latency Measurement 21-97
21.5.4 Performance Monitoring for Goldmont Plus Microarchitecture 21-98
21.5.4.1 Extended PEBS 21-98
21.5.5 Performance Monitoring for Tremont Microarchitecture 21-98
21.5.5.1 Adaptive PEBS 21-99
21.5.5.2 PEBS output to Intel® Processor Trace 21-99
21.5.5.3 Precise Distribution Support on Fixed Counter 0 21-101
21.5.5.4 Compatibility Enhancements to Offcore Response MSRs 21-101
21.6 PERFORMANCE MONITORING (LEGACY INTEL PROCESSORS) 21-103
21.6.1 Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors) 21-103
21.6.2 Performance Monitoring (Processors Based on Intel® Core™ Microarchitecture) 21-104
21.6.2.1 Fixed-function Performance Counters 21-105
21.6.2.2 Global Counter Control Facilities 21-106
21.6.2.3 At-Retirement Events 21-108

PAGE

21.6.2.4 Processor Event Based Sampling (PEBS) ... 21-108
21.6.3 Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture) ... 21-111
21.6.3.1 ESCR MSRs ... 21-114
21.6.3.2 Performance Counters ... 21-115
21.6.3.3 CCCR MSRs ... 21-116
21.6.3.4 Debug Store (DS) Mechanism ... 21-118
21.6.3.5 Programming the Performance Counters for Non-Retirement Events ... 21-118
21.6.3.6 At-Retirement Counting ... 21-124
21.6.3.7 Tagging Mechanism for Replay_event ... 21-125
21.6.3.8 Processor Event-Based Sampling (PEBS) ... 21-126
21.6.3.9 Operating System Implications ... 21-127
21.6.4 Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture ... 21-127
21.6.4.1 ESCR MSRs ... 21-127
21.6.4.2 CCCR MSRs ... 21-128
21.6.4.3 IA32_PEBS_ENABLE MSR ... 21-130
21.6.4.4 Performance Monitoring Events ... 21-130
21.6.4.5 Counting Clocks on systems with Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture ... 21-131
21.6.5 Performance Monitoring and Dual-Core Technology ... 21-132
21.6.6 Performance Monitoring on 64-bit Intel® Xeon® Processor MP with Up to 8-MByte L3 Cache ... 21-132
21.6.7 Performance Monitoring on L3 and Caching Bus Controller Sub-Systems ... 21-134
21.6.7.1 Overview of Performance Monitoring with L3/Caching Bus Controller ... 21-136
21.6.7.2 GBSQ Event Interface ... 21-137
21.6.7.3 GSNPQ Event Interface ... 21-138
21.6.7.4 FSB Event Interface ... 21-139
21.6.7.5 Common Event Control Interface ... 21-140
21.6.8 Performance Monitoring (P6 Family Processor) ... 21-140
21.6.8.1 PerfEvtSel0 and PerfEvtSel1 MSRs ... 21-141
21.6.8.2 PerfCtr0 and PerfCtr1 MSRs ... 21-142
21.6.8.3 Starting and Stopping the Performance-Monitoring Counters ... 21-142
21.6.8.4 Event and Time-Stamp Monitoring Software ... 21-142
21.6.8.5 Monitoring Counter Overflow ... 21-143
21.6.9 Performance Monitoring (Pentium Processors) ... 21-143
21.6.9.1 Control and Event Select Register (CESR) ... 21-144
21.6.9.2 Use of the Performance-Monitoring Pins ... 21-144
21.6.9.3 Events Counted ... 21-145
21.7 COUNTING CLOCKS ... 21-145
21.7.1 Non-Halted Reference Clockticks ... 21-146
21.7.2 Cycle Counting and Opportunistic Processor Operation ... 21-146
21.7.3 Determining the Processor Base Frequency ... 21-147
21.7.3.1 For Intel® Processors Based on Sandy Bridge, Ivy Bridge, Haswell, and Broadwell Microarchitectures ... 21-147
21.7.3.2 For Intel® Processors Based on Nehalem Microarchitecture ... 21-147
21.7.3.3 For Intel Atom® Processors Based on Silvermont Microarchitecture (Including Intel Processors Based on Airmont Microarchitecture) ... 21-147
21.7.3.4 For Intel® Core™ 2 Processor Family and for Intel® Xeon® Processors Based on Intel Core Microarchitecture ... 21-147
21.8 IA32_PERF_CAPABILITIES MSR ENUMERATION ... 21-148
21.8.1 Filtering of SMM Handler Overhead ... 21-149
21.9 PEBS FACILITY ... 21-149
21.9.1 Extended PEBS ... 21-149
21.9.2 Adaptive PEBS ... 21-151
21.9.2.1 Adaptive_Record Counter Control ... 21-152
21.9.2.2 PEBS Record Format ... 21-153
21.9.2.3 MSR_PEBS_DATA_CFG ... 21-157
21.9.2.4 PEBS Record Examples ... 21-160
21.9.3 Precise Distribution of Instructions Retired (PDIR) Facility ... 21-162
21.9.4 Reduced Skid PEBS ... 21-162
21.9.5 EPT-Friendly PEBS ... 21-163
21.9.6 PDist: Precise Distribution ... 21-163
21.9.7 Load Latency Facility ... 21-163
21.9.8 Store Latency Facility ... 21-165
21.9.9 Timed Processor Event Based Sampling ... 21-166
21.9.10 Counters Snapshotting ... 21-166
21.10 AUTO COUNTER RELOAD ... 21-166
21.10.1 Discovery and Interface ... 21-166

21.10.2 Configuration and Behavior 21-167
21.10.2.1 Reload Precision 21-167
21.10.2.2 PEBS Interaction 21-167
21.10.2.3 Precise Distribution (PDIST) Interaction 21-167

CHAPTER 22
8086 EMULATION

22.1 REAL-ADDRESS MODE 22-1
22.1.1 Address Translation in Real-Address Mode 22-2
22.1.2 Registers Supported in Real-Address Mode 22-3
22.1.3 Instructions Supported in Real-Address Mode 22-3
22.1.4 Interrupt and Exception Handling 22-4
22.2 VIRTUAL-8086 MODE 22-5
22.2.1 Enabling Virtual-8086 Mode 22-6
22.2.2 Structure of a Virtual-8086 Task 22-7
22.2.3 Paging of Virtual-8086 Tasks 22-7
22.2.4 Protection within a Virtual-8086 Task 22-8
22.2.5 Entering Virtual-8086 Mode 22-8
22.2.6 Leaving Virtual-8086 Mode 22-9
22.2.7 Sensitive Instructions 22-10
22.2.8 Virtual-8086 Mode I/O 22-10
22.2.8.1 I/O-Port-Mapped I/O 22-11
22.2.8.2 Memory-Mapped I/O 22-11
22.2.8.3 Special I/O Buffers 22-11
22.3 INTERRUPT AND EXCEPTION HANDLING IN VIRTUAL-8086 MODE 22-11
22.3.1 Class 1—Hardware Interrupt and Exception Handling in Virtual-8086 Mode 22-12
22.3.1.1 Handling an Interrupt or Exception Through a Protected-Mode Trap or Interrupt Gate 22-12
22.3.1.2 Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler 22-14
22.3.1.3 Handling an Interrupt or Exception Through a Task Gate 22-14
22.3.2 Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism 22-15
22.3.3 Class 3—Software Interrupt Handling in Virtual-8086 Mode 22-16
22.3.3.1 Method 1: Software Interrupt Handling 22-18
22.3.3.2 Methods 2 and 3: Software Interrupt Handling 22-18
22.3.3.3 Method 4: Software Interrupt Handling 22-19
22.3.3.4 Method 5: Software Interrupt Handling 22-19
22.3.3.5 Method 6: Software Interrupt Handling 22-19
22.4 PROTECTED-MODE VIRTUAL INTERRUPTS 22-20

CHAPTER 23
MIXING 16-BIT AND 32-BIT CODE

23.1 DEFINING 16-BIT AND 32-BIT PROGRAM MODULES 23-1
23.2 MIXING 16-BIT AND 32-BIT OPERATIONS WITHIN A CODE SEGMENT 23-2
23.3 SHARING DATA AMONG MIXED-SIZE CODE SEGMENTS 23-3
23.4 TRANSFERRING CONTROL AMONG MIXED-SIZE CODE SEGMENTS 23-3
23.4.1 Code-Segment Pointer Size 23-4
23.4.2 Stack Management for Control Transfer 23-4
23.4.2.1 Controlling the Operand-Size Attribute For a Call 23-5
23.4.2.2 Passing Parameters With a Gate 23-6
23.4.3 Interrupt Control Transfers 23-6
23.4.4 Parameter Translation 23-6
23.4.5 Writing Interface Procedures 23-6

CHAPTER 24
ARCHITECTURE COMPATIBILITY

24.1 PROCESSOR FAMILIES AND CATEGORIES 24-1
24.2 RESERVED BITS 24-2
24.3 ENABLING NEW FUNCTIONS AND MODES 24-2
24.4 DETECTING THE PRESENCE OF NEW FEATURES THROUGH SOFTWARE 24-2
24.5 INTEL MMX TECHNOLOGY 24-2
24.6 STREAMING SIMD EXTENSIONS (SSE) 24-3
24.7 STREAMING SIMD EXTENSIONS 2 (SSE2) 24-3

PAGE

24.8 STREAMING SIMD EXTENSIONS 3 (SSE3) 24-3
24.9 ADDITIONAL STREAMING SIMD EXTENSIONS 24-3
24.10 INTEL HYPER-THREADING TECHNOLOGY 24-3
24.11 MULTI-CORE TECHNOLOGY 24-4
24.12 SPECIFIC FEATURES OF DUAL-CORE PROCESSOR 24-4
24.13 NEW INSTRUCTIONS IN THE PENTIUM AND LATER IA-32 PROCESSORS 24-4
24.13.1 Instructions Added Prior to the Pentium Processor 24-4
24.14 OBSOLETE INSTRUCTIONS 24-5
24.15 UNDEFINED OPCODES 24-5
24.16 NEW FLAGS IN THE EFLAGS REGISTER 24-6
24.16.1 Using EFLAGS Flags to Distinguish Between 32-Bit IA-32 Processors 24-6
24.17 STACK OPERATIONS AND USER SOFTWARE 24-7
24.17.1 PUSH SP 24-7
24.17.2 EFLAGS Pushed on the Stack 24-7
24.18 X87 FPU 24-7
24.18.1 Control Register CR0 Flags 24-8
24.18.2 x87 FPU Status Word 24-8
24.18.2.1 Condition Code Flags (C0 through C3) 24-8
24.18.2.2 Stack Fault Flag 24-8
24.18.3 x87 FPU Control Word 24-9
24.18.4 x87 FPU Tag Word 24-9
24.18.5 Data Types 24-9
24.18.5.1 NaNs 24-9
24.18.5.2 Pseudo-zero, Pseudo-NaN, Pseudo-infinity, and Unnormal Formats 24-9
24.18.6 Floating-Point Exceptions 24-10
24.18.6.1 Denormal Operand Exception (#D) 24-10
24.18.6.2 Numeric Overflow Exception (#O) 24-10
24.18.6.3 Numeric Underflow Exception (#U) 24-10
24.18.6.4 Exception Precedence 24-10
24.18.6.5 CS and EIP For FPU Exceptions 24-11
24.18.6.6 FPU Error Signals 24-11
24.18.6.7 Assertion of the FERR# Pin 24-11
24.18.6.8 Invalid Operation Exception On Denormals 24-11
24.18.6.9 Alignment Check Exceptions (#AC) 24-11
24.18.6.10 Segment Not Present Exception During FLDENV 24-12
24.18.6.11 Device Not Available Exception (#NM) 24-12
24.18.6.12 Coprocessor Segment Overrun Exception 24-12
24.18.6.13 General Protection Exception (#GP) 24-12
24.18.6.14 Floating-Point Error Exception (#MF) 24-12
24.18.7 Changes to Floating-Point Instructions 24-12
24.18.7.1 FDIV, FPREM, and FSQRT Instructions 24-12
24.18.7.2 FSCALE Instruction 24-12
24.18.7.3 FPREM1 Instruction 24-13
24.18.7.4 FPREM Instruction 24-13
24.18.7.5 FUCOM, FUCOMP, and FUCOMPP Instructions 24-13
24.18.7.6 FPTAN Instruction 24-13
24.18.7.7 Stack Overflow 24-13
24.18.7.8 FSIN, FCOS, and FSINCOS Instructions 24-13
24.18.7.9 FPATAN Instruction 24-13
24.18.7.10 F2XM1 Instruction 24-13
24.18.7.11 FLD Instruction 24-14
24.18.7.12 FXTRACT Instruction 24-14
24.18.7.13 Load Constant Instructions 24-14
24.18.7.14 FXAM Instruction 24-14
24.18.7.15 FSAVE and FSTENV Instructions 24-14
24.18.8 Transcendental Instructions 24-14
24.18.9 Obsolete Instructions and Undefined Opcodes 24-15
24.18.10 WAIT/FWAIT Prefix Differences 24-15
24.18.11 Operands Split Across Segments and/or Pages 24-15
24.18.12 FPU Instruction Synchronization 24-15
24.19 SERIALIZING INSTRUCTIONS 24-16
24.20 FPU AND MATH COPROCESSOR INITIALIZATION 24-16
24.20.1 Intel® 387 and Intel® 287 Math Coprocessor Initialization 24-16
24.20.2 Intel486 SX Processor and Intel 487 SX Math Coprocessor Initialization 24-16
24.21 CONTROL REGISTERS 24-17

PAGE

24.22 MEMORY MANAGEMENT FACILITIES 24-19
24.22.1 New Memory Management Control Flags 24-19
24.22.1.1 Physical Memory Addressing Extension 24-19
24.22.1.2 Global Pages 24-19
24.22.1.3 Larger Page Sizes 24-19
24.22.2 CD and NW Cache Control Flags 24-19
24.22.3 Descriptor Types and Contents 24-19
24.22.4 Changes in Segment Descriptor Loads 24-20
24.23 DEBUG FACILITIES 24-20
24.23.1 Differences in Debug Register DR6 24-20
24.23.2 Differences in Debug Register DR7 24-20
24.23.3 Debug Registers DR4 and DR5 24-20
24.24 RECOGNITION OF BREAKPOINTS 24-20
24.25 EXCEPTIONS AND/OR EXCEPTION CONDITIONS 24-21
24.25.1 Machine-Check Architecture 24-22
24.25.2 Priority of Exceptions 24-22
24.25.3 Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers 24-22
24.26 INTERRUPTS 24-27
24.26.1 Interrupt Propagation Delay 24-27
24.26.2 NMI Interrupts 24-27
24.26.3 IDT Limit 24-27
24.27 ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC) 24-27
24.27.1 Software Visible Differences Between the Local APIC and the 82489DX 24-28
24.27.2 New Features Incorporated in the Local APIC for the P6 Family and Pentium Processors 24-28
24.27.3 New Features Incorporated in the Local APIC of the Pentium 4 and Intel Xeon Processors 24-28
24.28 TASK SWITCHING AND TSS 24-28
24.28.1 P6 Family and Pentium Processor TSS 24-29
24.28.2 TSS Selector Writes 24-29
24.28.3 Order of Reads/Writes to the TSS 24-29
24.28.4 Using A 16-Bit TSS with 32-Bit Constructs 24-29
24.28.5 Differences in I/O Map Base Addresses 24-29
24.29 CACHE MANAGEMENT 24-30
24.29.1 Self-Modifying Code with Cache Enabled 24-30
24.29.2 Disabling the L3 Cache 24-31
24.30 PAGING 24-31
24.30.1 Large Pages 24-31
24.30.2 PCD and PWT Flags 24-31
24.30.3 Enabling and Disabling Paging 24-32
24.31 STACK OPERATIONS AND SUPERVISOR SOFTWARE 24-32
24.31.1 Selector Pushes and Pops 24-32
24.31.2 Error Code Pushes 24-32
24.31.3 Fault Handling Effects on the Stack 24-33
24.31.4 Interlevel RET/IRET From a 16-Bit Interrupt or Call Gate 24-33
24.32 MIXING 16- AND 32-BIT SEGMENTS 24-33
24.33 SEGMENT AND ADDRESS WRAPAROUND 24-33
24.33.1 Segment Wraparound 24-34
24.34 STORE BUFFERS AND MEMORY ORDERING 24-34
24.35 BUS LOCKING 24-35
24.36 BUS HOLD 24-35
24.37 MODEL-SPECIFIC EXTENSIONS TO THE IA-32 24-35
24.37.1 Model-Specific Registers 24-36
24.37.2 RDMSR and WRMSR Instructions 24-36
24.37.3 Memory Type Range Registers 24-36
24.37.4 Machine-Check Exception and Architecture 24-36
24.37.5 Performance-Monitoring Counters 24-37
24.38 TWO WAYS TO RUN INTEL 286 PROCESSOR TASKS 24-37
24.39 INITIAL STATE OF PENTIUM, PENTIUM PRO AND PENTIUM 4 PROCESSORS 24-37

CHAPTER 25
INTRODUCTION TO VIRTUAL MACHINE EXTENSIONS

25.1 OVERVIEW 25-1
25.2 VIRTUAL MACHINE ARCHITECTURE 25-1
25.3 INTRODUCTION TO VMX OPERATION 25-1
25.4 LIFE CYCLE OF VMM SOFTWARE 25-2

PAGE

25.5 VIRTUAL-MACHINE CONTROL STRUCTURE 25-2
25.6 DISCOVERING SUPPORT FOR VMX 25-2
25.7 ENABLING AND ENTERING VMX OPERATION 25-3
25.8 RESTRICTIONS ON VMX OPERATION 25-3

CHAPTER 26
VIRTUAL MACHINE CONTROL STRUCTURES

26.1 OVERVIEW 26-1
26.2 FORMAT OF THE VMCS REGION 26-2
26.3 ORGANIZATION OF VMCS DATA 26-3
26.4 GUEST-STATE AREA 26-4
26.4.1 Guest Register State 26-4
26.4.2 Guest Non-Register State 26-6
26.5 HOST-STATE AREA 26-8
26.6 VM-EXECUTION CONTROL FIELDS 26-9
26.6.1 Pin-Based VM-Execution Controls 26-9
26.6.2 Processor-Based VM-Execution Controls 26-10
26.6.3 Exception Bitmap 26-14
26.6.4 I/O-Bitmap Addresses 26-14
26.6.5 Time-Stamp Counter Offset and Multiplier 26-14
26.6.6 Guest/Host Masks and Read Shadows for CR0 and CR4 26-14
26.6.7 CR3-Target Controls 26-14
26.6.8 Controls for APIC Virtualization 26-15
26.6.9 MSR-Bitmap Address 26-16
26.6.10 Executive-VMCS Pointer 26-16
26.6.11 Extended-Page-Table Pointer (EPTP) 26-17
26.6.12 Virtual-Processor Identifier (VPID) 26-17
26.6.13 Controls for PAUSE-Loop Exiting 26-17
26.6.14 VM-Function Controls 26-18
26.6.15 VMCS Shadowing Bitmap Addresses 26-18
26.6.16 ENCLS-Exiting Bitmap 26-18
26.6.17 ENCLV-Exiting Bitmap 26-18
26.6.18 PCONFIG-Exiting Bitmap 26-18
26.6.19 Control Field for Page-Modification Logging 26-19
26.6.20 Controls for Virtualization Exceptions 26-19
26.6.21 XSS-Exiting Bitmap 26-19
26.6.22 Sub-Page-Permission-Table Pointer (SPPTP) 26-19
26.6.23 Fields Related to Hypervisor-Managed Linear-Address Translation 26-20
26.6.24 Fields Related to PASID Translation 26-20
26.6.25 Instruction-Timeout Control 26-20
26.6.26 Fields Controlling Virtualization of the IA32_SPEC_CTRL MSR 26-21
26.7 VM-EXIT CONTROL FIELDS 26-21
26.7.1 VM-Exit Controls 26-21
26.7.2 VM-Exit Controls for MSRs 26-22
26.8 VM-ENTRY CONTROL FIELDS 26-23
26.8.1 VM-Entry Controls 26-23
26.8.2 VM-Entry Controls for MSRs 26-24
26.8.3 VM-Entry Controls for Event Injection 26-24
26.9 VM-EXIT INFORMATION FIELDS 26-25
26.9.1 Basic VM-Exit Information 26-26
26.9.2 Information for VM Exits Due to Vectored Events 26-27
26.9.3 Information for VM Exits That Occur During Event Delivery 26-27
26.9.4 Information for VM Exits Due to Instruction Execution 26-28
26.9.5 VM-Instruction Error Field 26-29
26.10 VMCS TYPES: ORDINARY AND SHADOW 26-29
26.11 SOFTWARE USE OF THE VMCS AND RELATED STRUCTURES 26-29
26.11.1 Software Use of Virtual-Machine Control Structures 26-29
26.11.2 VMREAD, VMWRITE, and Encodings of VMCS Fields 26-30
26.11.3 Initializing a VMCS 26-32
26.11.4 Software Access to Related Structures 26-32
26.11.5 VMXON Region 26-32

CHAPTER 27
VMX NON-ROOT OPERATION

27.1 INSTRUCTIONS THAT CAUSE VM EXITS 27-1
27.1.1 Relative Priority of Faults and VM Exits 27-1
27.1.2 Instructions That Cause VM Exits Unconditionally 27-2
27.1.3 Instructions That Cause VM Exits Conditionally 27-2
27.2 OTHER CAUSES OF VM EXITS 27-6
27.3 CHANGES TO INSTRUCTION BEHAVIOR IN VMX NON-ROOT OPERATION 27-8
27.4 OTHER CHANGES IN VMX NON-ROOT OPERATION 27-14
27.4.1 Event Blocking 27-14
27.4.2 Treatment of Task Switches 27-14
27.4.3 Shadow-Stack Updates 27-15
27.5 FEATURES SPECIFIC TO VMX NON-ROOT OPERATION 27-15
27.5.1 VMX-Preemption Timer 27-15
27.5.2 Monitor Trap Flag 27-16
27.5.3 Translation of Guest-Physical Addresses Using EPT 27-17
27.5.4 Translation of Guest-Physical Addresses Used by Intel Processor Trace 27-17
27.5.4.1 Guest-Physical Address Translation for Intel PT: Details 27-17
27.5.4.2 Trace-Address Pre-Translation (TAPT) 27-18
27.5.5 APIC Virtualization 27-18
27.5.6 VM Functions 27-18
27.5.6.1 Enabling VM Functions 27-18
27.5.6.2 General Operation of the VMFUNC Instruction 27-19
27.5.6.3 EPTP Switching 27-19
27.5.7 Virtualization Exceptions 27-20
27.5.7.1 Convertible EPT Violations 27-21
27.5.7.2 Virtualization-Exception Information 27-21
27.5.7.3 Delivery of Virtualization Exceptions 27-22
27.5.8 PASID Translation 27-22
27.6 UNRESTRICTED GUESTS 27-23

CHAPTER 28
VM ENTRIES

28.1 BASIC VM-ENTRY CHECKS 28-2
28.2 CHECKS ON VMX CONTROLS AND HOST-STATE AREA 28-2
28.2.1 Checks on VMX Controls 28-2
28.2.1.1 VM-Execution Control Fields 28-2
28.2.1.2 VM-Exit Control Fields 28-5
28.2.1.3 VM-Entry Control Fields 28-6
28.2.2 Checks on Host Control Registers, MSRs, and SSP 28-7
28.2.3 Checks on Host Segment and Descriptor-Table Registers 28-7
28.2.4 Checks Related to Address-Space Size 28-8
28.3 CHECKING AND LOADING GUEST STATE 28-8
28.3.1 Checks on the Guest State Area 28-8
28.3.1.1 Checks on Guest Control Registers, Debug Registers, and MSRs 28-9
28.3.1.2 Checks on Guest Segment Registers 28-10
28.3.1.3 Checks on Guest Descriptor-Table Registers 28-12
28.3.1.4 Checks on Guest RIP, RFLAGS, and SSP 28-12
28.3.1.5 Checks on Guest Non-Register State 28-13
28.3.1.6 Checks on Guest Page-Directory-Pointer-Table Entries 28-15
28.3.2 Loading Guest State 28-15
28.3.2.1 Loading Guest Control Registers, Debug Registers, and MSRs 28-15
28.3.2.2 Loading Guest Segment Registers and Descriptor-Table Registers 28-17
28.3.2.3 Loading Guest RIP, RSP, RFLAGS, and SSP 28-17
28.3.2.4 Loading Page-Directory-Pointer-Table Entries 28-18
28.3.2.5 Updating Non-Register State 28-18
28.3.3 Clearing Address-Range Monitoring 28-18
28.4 LOADING MSRS 28-18
28.5 TRACE-ADDRESS PRE-TRANSLATION (TAPT) 28-19
28.6 EVENT INJECTION 28-19
28.6.1 Vectored-Event Injection 28-19
28.6.1.1 Details of Vectored-Event Injection 28-20
28.6.1.2 VM Exits During Event Injection 28-21

PAGE

28.6.1.3 Event Injection for VM Entries to Real-Address Mode 28-22
28.6.2 Injection of Pending MTF VM Exits 28-22
28.7 SPECIAL FEATURES OF VM ENTRY 28-22
28.7.1 Interruptibility State 28-23
28.7.2 Activity State 28-23
28.7.3 Delivery of Pending Debug Exceptions after VM Entry 28-24
28.7.4 VMX-Preemption Timer 28-25
28.7.5 Interrupt-Window Exiting and Virtual-Interrupt Delivery 28-25
28.7.6 NMI-Window Exiting 28-25
28.7.7 VM Exits Induced by the TPR Threshold 28-25
28.7.8 Pending MTF VM Exits 28-26
28.7.9 VM Entries and Advanced Debugging Features 28-26
28.7.10 User-Interrupt Recognition After VM Entry 28-26
28.8 VM-ENTRY FAILURES DURING OR AFTER LOADING GUEST STATE 28-26
28.9 MACHINE-CHECK EVENTS DURING VM ENTRY 28-27

CHAPTER 29
VM EXITS

29.1 ARCHITECTURAL STATE BEFORE A VM EXIT 29-1
29.2 RECORDING VM-EXIT INFORMATION AND UPDATING VM-ENTRY CONTROL FIELDS 29-4
29.2.1 Basic VM-Exit Information 29-4
29.2.2 Information for VM Exits Due to Vectored Events 29-12
29.2.3 Information About NMI Unblocking Due to IRET 29-13
29.2.4 Information for VM Exits During Event Delivery 29-14
29.2.5 Information for VM Exits Due to Instruction Execution 29-15
29.3 SAVING GUEST STATE 29-23
29.3.1 Saving Control Registers, Debug Registers, and MSRs 29-23
29.3.2 Saving Segment Registers and Descriptor-Table Registers 29-24
29.3.3 Saving RIP, RSP, RFLAGS, and SSP 29-24
29.3.4 Saving Non-Register State 29-26
29.4 SAVING MSRS 29-28
29.5 LOADING HOST STATE 29-28
29.5.1 Loading Host Control Registers, Debug Registers, MSRs 29-29
29.5.2 Loading Host Segment and Descriptor-Table Registers 29-30
29.5.3 Loading Host RIP, RSP, RFLAGS, and SSP 29-31
29.5.4 Checking and Loading Host Page-Directory-Pointer-Table Entries 29-31
29.5.5 Updating Non-Register State 29-32
29.5.6 Clearing Address-Range Monitoring 29-32
29.6 LOADING MSRS 29-32
29.7 VMX ABORTS 29-33
29.8 MACHINE-CHECK EVENTS DURING VM EXIT 29-33
29.9 USER-INTERRUPT RECOGNITION AFTER VM EXIT 29-34

CHAPTER 30
VMX SUPPORT FOR ADDRESS TRANSLATION

30.1 VIRTUAL PROCESSOR IDENTIFIERS (VPIDS) 30-1
30.2 HYPERVISOR-MANAGED LINEAR-ADDRESS TRANSLATION (HLAT) 30-1
30.3 THE EXTENDED PAGE TABLE MECHANISM (EPT) 30-1
30.3.1 EPT Overview 30-2
30.3.2 EPT Translation Mechanism 30-3
30.3.3 EPT-Induced VM Exits 30-10
30.3.3.1 EPT Misconfigurations 30-10
30.3.3.2 EPT Violations 30-12
30.3.3.3 Prioritization of EPT Misconfigurations and EPT Violations 30-15
30.3.4 Sub-Page Write Permissions 30-16
30.3.4.1 Write Accesses That Are Eligible for Sub-Page Write Permissions 30-17
30.3.4.2 Determining an Access's Sub-Page Write Permission 30-17
30.3.5 Accessed and Dirty Flags for EPT 30-18
30.3.6 Page-Modification Logging 30-19
30.3.7 EPT and Memory Typing 30-19
30.3.7.1 Memory Type Used for Accessing EPT Paging Structures 30-19
30.3.7.2 Memory Type Used for Translated Guest-Physical Addresses 30-19

30.4 CACHING TRANSLATION INFORMATION 30-20
30.4.1 Information That May Be Cached 30-20
30.4.2 Creating and Using Cached Translation Information 30-21
30.4.3 Invalidating Cached Translation Information 30-22
30.4.3.1 Operations that Invalidate Cached Mappings 30-22
30.4.3.2 Operations that Need Not Invalidate Cached Mappings 30-24
30.4.3.3 Guidelines for Use of the INVVPID Instruction 30-24
30.4.3.4 Guidelines for Use of the INVEPT Instruction 30-25

CHAPTER 31
APIC VIRTUALIZATION AND VIRTUAL INTERRUPTS

31.1 VIRTUAL APIC STATE 31-1
31.1.1 Virtualized APIC Registers 31-2
31.1.2 TPR Virtualization 31-2
31.1.3 PPR Virtualization 31-2
31.1.4 EOI Virtualization 31-3
31.1.5 Self-IPI Virtualization 31-3
31.1.6 IPI Virtualization 31-3
31.1.6.1 Virtual-Interrupt Posting 31-4
31.1.6.2 IPI Virtualization Using Virtual-Interrupt Posting 31-4
31.2 EVALUATION AND DELIVERY OF VIRTUAL INTERRUPTS 31-5
31.2.1 Evaluation of Pending Virtual Interrupts 31-5
31.2.2 Virtual-Interrupt Delivery 31-6
31.2.3 Virtualizing User-Interrupt Notifications 31-6
31.3 VIRTUALIZING CR8-BASED TPR ACCESSES 31-7
31.4 VIRTUALIZING MEMORY-MAPPED APIC ACCESSES 31-7
31.4.1 Priority of APIC-Access VM Exits 31-8
31.4.2 Virtualizing Reads from the APIC-Access Page 31-9
31.4.3 Virtualizing Writes to the APIC-Access Page 31-10
31.4.3.1 Determining Whether a Write Access is Virtualized 31-10
31.4.3.2 APIC-Write Emulation 31-11
31.4.3.3 APIC-Write VM Exits 31-12
31.4.4 Instruction-Specific Considerations 31-12
31.4.5 Issues Pertaining to Page Size and TLB Management 31-13
31.4.6 APIC Accesses Not Directly Resulting From Linear Addresses 31-13
31.4.6.1 Guest-Physical Accesses to the APIC-Access Page 31-14
31.4.6.2 Physical Accesses to the APIC-Access Page 31-14
31.5 VIRTUALIZING MSR-BASED APIC ACCESSES 31-15
31.6 POSTED-INTERRUPT PROCESSING 31-16
31.7 VIRTUALIZING SENDUIPI 31-17

CHAPTER 32
VMX INSTRUCTION REFERENCE

32.1 OVERVIEW 32-1
32.2 CONVENTIONS 32-2
32.3 VMX INSTRUCTIONS 32-2
INVEPT— Invalidate Translations Derived from EPT 32-3
INVVPID— Invalidate Translations Based on VPID 32-6
VMCALL—Call to VM Monitor 32-9
VMCLEAR—Clear Virtual-Machine Control Structure 32-11
VMFUNC—Invoke VM function 32-13
VMLAUNCH/VMRESUME—Launch/Resume Virtual Machine 32-14
VMPTRLD—Load Pointer to Virtual-Machine Control Structure 32-17
VMPTRST—Store Pointer to Virtual-Machine Control Structure 32-19
VMREAD—Read Field from Virtual-Machine Control Structure 32-21
VMRESUME—Resume Virtual Machine 32-23
VMWRITE—Write Field to Virtual-Machine Control Structure 32-24
VMXOFF—Leave VMX Operation 32-26
VMXON—Enter VMX Operation 32-28
32.4 VM INSTRUCTION ERROR NUMBERS 32-31

PAGE

CHAPTER 33
SYSTEM MANAGEMENT MODE

33.1 SYSTEM MANAGEMENT MODE OVERVIEW 33-1
33.1.1 System Management Mode and VMX Operation 33-2
33.2 SYSTEM MANAGEMENT INTERRUPT (SMI) 33-2
33.3 SWITCHING BETWEEN SMM AND THE OTHER PROCESSOR OPERATING MODES 33-2
33.3.1 Entering SMM 33-2
33.3.2 Exiting From SMM 33-3
33.4 SMRAM 33-4
33.4.1 SMRAM State Save Map 33-4
33.4.1.1 SMRAM State Save Map and Intel 64 Architecture 33-6
33.4.2 SMRAM Caching 33-8
33.4.2.1 System Management Range Registers (SMRR) 33-9
33.5 SMI HANDLER EXECUTION ENVIRONMENT 33-9
33.5.1 Initial SMM Execution Environment 33-9
33.5.2 SMI Handler Operating Mode Switching 33-10
33.5.3 Control-flow Enforcement Technology Interactions 33-11
33.6 EXCEPTIONS AND INTERRUPTS WITHIN SMM 33-11
33.7 MANAGING SYNCHRONOUS AND ASYNCHRONOUS SYSTEM MANAGEMENT INTERRUPTS 33-12
33.7.1 I/O State Implementation 33-12
33.8 NMI HANDLING WHILE IN SMM 33-13
33.9 SMM REVISION IDENTIFIER 33-13
33.10 AUTO HALT RESTART 33-14
33.10.1 Executing the HLT Instruction in SMM 33-14
33.11 SMBASE RELOCATION 33-14
33.12 I/O INSTRUCTION RESTART 33-15
33.12.1 Back-to-Back SMI Interrupts When I/O Instruction Restart Is Being Used 33-16
33.13 SMM MULTIPLE-PROCESSOR CONSIDERATIONS 33-16
33.14 DEFAULT TREATMENT OF SMIS AND SMM WITH VMX OPERATION AND SMX OPERATION 33-16
33.14.1 Default Treatment of SMI Delivery 33-17
33.14.2 Default Treatment of RSM 33-18
33.14.3 Protection of CR4.VMXE in SMM 33-19
33.14.4 VMXOFF and SMI Unblocking 33-19
33.15 DUAL-MONITOR TREATMENT OF SMIs AND SMM 33-19
33.15.1 Dual-Monitor Treatment Overview 33-19
33.15.2 SMM VM Exits 33-20
33.15.2.1 Architectural State Before a VM Exit 33-20
33.15.2.2 Updating the Current-VMCS and Executive-VMCS Pointers 33-20
33.15.2.3 Recording VM-Exit Information 33-20
33.15.2.4 Saving Guest State 33-21
33.15.2.5 Updating State 33-21
33.15.3 Operation of the SMM-Transfer Monitor 33-22
33.15.4 VM Entries that Return from SMM 33-22
33.15.4.1 Checks on the Executive-VMCS Pointer Field 33-22
33.15.4.2 Checks on VM-Execution Control Fields 33-22
33.15.4.3 Checks on VM-Entry Control Fields 33-23
33.15.4.4 Checks on the Guest State Area 33-23
33.15.4.5 Loading Guest State 33-23
33.15.4.6 VMX-Preemption Timer 33-23
33.15.4.7 Updating the Current-VMCS and SMM-Transfer VMCS Pointers 33-24
33.15.4.8 VM Exits Induced by VM Entry 33-24
33.15.4.9 SMI Blocking 33-24
33.15.4.10 Failures of VM Entries That Return from SMM 33-24
33.15.5 Enabling the Dual-Monitor Treatment 33-25
33.15.6 Activating the Dual-Monitor Treatment 33-26
33.15.6.1 Initial Checks 33-26
33.15.6.2 Updating the Current-VMCS and Executive-VMCS Pointers 33-27
33.15.6.3 Saving Guest State 33-27
33.15.6.4 Saving MSRs 33-27
33.15.6.5 Loading Host State 33-27
33.15.6.6 Loading MSRs 33-29
33.15.7 Deactivating the Dual-Monitor Treatment 33-29
33.16 SMI AND PROCESSOR EXTENDED STATE MANAGEMENT 33-29
33.17 MODEL-SPECIFIC SYSTEM MANAGEMENT ENHANCEMENT 33-30

33.17.1 SMM Handler Code Access Control 33-30
33.17.2 SMI Delivery Delay Reporting 33-30
33.17.3 Blocked SMI Reporting 33-30

CHAPTER 34
INTEL® PROCESSOR TRACE

34.1 OVERVIEW 34-1
34.1.1 Features and Capabilities 34-1
34.1.1.1 Packet Summary 34-1
34.2 INTEL® PROCESSOR TRACE OPERATIONAL MODEL 34-2
34.2.1 Change of Flow Instruction (COFI) Tracing 34-2
34.2.1.1 Direct Transfer COFI 34-3
34.2.1.2 Indirect Transfer COFI 34-3
34.2.1.3 Far Transfer COFI 34-4
34.2.2 Software Trace Instrumentation with PTWRITE 34-4
34.2.3 Power Event Tracing 34-4
34.2.4 Event Tracing 34-5
34.2.5 Trace Filtering 34-5
34.2.5.1 Filtering by Current Privilege Level (CPL) 34-5
34.2.5.2 Filtering by CR3 34-5
34.2.5.3 Filtering by IP 34-6
34.2.6 Packet Generation Enable Controls 34-7
34.2.6.1 Packet Enable (PacketEn) 34-7
34.2.6.2 Trigger Enable (TriggerEn) 34-8
34.2.6.3 Context Enable (ContextEn) 34-8
34.2.6.4 Branch Enable (BranchEn) 34-8
34.2.6.5 Filter Enable (FilterEn) 34-8
34.2.7 Trace Output 34-9
34.2.7.1 Single Range Output 34-9
34.2.7.2 Table of Physical Addresses (ToPA) 34-10
Single Output Region ToPA Implementation 34-12
ToPA Table Entry Format 34-12
ToPA STOP 34-13
ToPA PMI 34-13
PMI Preservation 34-14
ToPA PMI and Single Output Region ToPA Implementation 34-14
ToPA PMI and XSAVES/XRSTORS State Handling 34-15
ToPA Errors 34-15
34.2.7.3 Trace Transport Subsystem 34-16
34.2.7.4 Restricted Memory Access 34-16
Modifications to Restricted Memory Regions 34-16
34.2.8 Enabling and Configuration MSRs 34-17
34.2.8.1 General Considerations 34-17
34.2.8.2 IA32_RTIT_CTL MSR 34-17
34.2.8.3 Enabling and Disabling Packet Generation with TraceEn 34-20
Disabling Packet Generation 34-21
Other Writes to IA32_RTIT_CTL 34-21
34.2.8.4 IA32_RTIT_STATUS MSR 34-21
34.2.8.5 IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B MSRs 34-22
34.2.8.6 IA32_RTIT_CR3_MATCH MSR 34-22
34.2.8.7 IA32_RTIT_OUTPUT_BASE MSR 34-23
34.2.8.8 IA32_RTIT_OUTPUT_MASK_PTRS MSR 34-23
34.2.9 Interaction of Intel® Processor Trace and Other Processor Features 34-24
34.2.9.1 Intel® Transactional Synchronization Extensions (Intel® TSX) 34-24
34.2.9.2 TSX and IP Filtering 34-25
34.2.9.3 System Management Mode (SMM) 34-25
34.2.9.4 Virtual-Machine Extensions (VMX) 34-26
34.2.9.5 Intel® Software Guard Extensions (Intel® SGX) 34-26
34.2.9.6 SENTER/ENTERACCS and ACM 34-26
34.2.9.7 Intel® Memory Protection Extensions (Intel® MPX) 34-26
34.3 CONFIGURATION AND PROGRAMMING GUIDELINE 34-26
34.3.1 Detection of Intel Processor Trace and Capability Enumeration 34-26

PAGE

34.3.1.1 Packet Decoding of RIP versus LIP 34-30
34.3.1.2 Model Specific Capability Restrictions 34-30
34.3.2 Enabling and Configuration of Trace Packet Generation 34-30
34.3.2.1 Enabling Packet Generation 34-30
34.3.2.2 Disabling Packet Generation 34-31
34.3.3 Flushing Trace Output 34-31
34.3.4 Warm Reset 34-31
34.3.5 Context Switch Consideration 34-31
34.3.5.1 Manual Trace Configuration Context Switch 34-31
34.3.5.2 Trace Configuration Context Switch Using XSAVES/XRSTORS 34-32
34.3.6 Cycle-Accurate Mode 34-32
34.3.6.1 Cycle Counter 34-33
34.3.6.2 Cycle Packet Semantics 34-33
34.3.6.3 Cycle Thresholds 34-33
34.3.7 Decoder Synchronization (PSB+) 34-34
34.3.8 Internal Buffer Overflow 34-35
34.3.8.1 Overflow Impact on Enables 34-35
34.3.8.2 Overflow Impact on Timing Packets 34-35
34.3.9 TNT Disable 34-36
34.3.10 Operational Errors 34-36
34.4 TRACE PACKETS AND DATA TYPES 34-36
34.4.1 Packet Relationships and Ordering 34-36
34.4.1.1 Packet Blocks 34-37
Decoder Implications 34-37
34.4.2 Packet Definitions 34-38
34.4.2.1 Taken/Not-taken (TNT) Packet 34-39
34.4.2.2 Target IP (TIP) Packet 34-40
IP Compression 34-40
Indirect Transfer Compression for Returns (RET) 34-41
34.4.2.3 Deferred TIPs 34-42
34.4.2.4 Packet Generation Enable (TIP.PGE) Packet 34-43
34.4.2.5 Packet Generation Disable (TIP.PGD) Packet 34-44
34.4.2.6 Flow Update (FUP) Packet 34-45
FUP IP Payload 34-45
34.4.2.7 Paging Information (PIP) Packet 34-47
34.4.2.8 MODE Packets 34-47
MODE.Exec Packet 34-48
MODE.TSX Packet 34-49
34.4.2.9 TraceStop Packet 34-50
34.4.2.10 Core:Bus Ratio (CBR) Packet 34-50
34.4.2.11 Timestamp Counter (TSC) Packet 34-51
34.4.2.12 Mini Time Counter (MTC) Packet 34-52
34.4.2.13 TSC/MTC Alignment (TMA) Packet 34-53
34.4.2.14 Cycle Count (CYC) Packet 34-54
34.4.2.15 VMCS Packet 34-55
34.4.2.16 Overflow (OVF) Packet 34-56
34.4.2.17 Packet Stream Boundary (PSB) Packet 34-56
34.4.2.18 PSBEND Packet 34-57
34.4.2.19 Maintenance (MNT) Packet 34-58
34.4.2.20 PAD Packet 34-58
34.4.2.21 PTWRITE (PTW) Packet 34-59
34.4.2.22 Execution Stop (EXSTOP) Packet 34-60
34.4.2.23 MWAIT Packet 34-61
34.4.2.24 Power Entry (PWRE) Packet 34-62
34.4.2.25 Power Exit (PWRX) Packet 34-63
34.4.2.26 Block Begin Packet (BBP) 34-64
34.4.2.27 Block Item Packet (BIP) 34-65
BIP State Value Encodings 34-65
34.4.2.28 Block End Packet (BEP) 34-70
34.4.2.29 Control Flow Event (CFE) Packet 34-71
CFE Packet Type and Vector Fields 34-71
34.4.2.30 Event Data (EVD) Packet 34-73
34.5 TRACING IN VMX OPERATION 34-73
34.5.1 VMX-Specific Packets and VMCS Controls 34-74

34.5.2 Managing Trace Packet Generation Across VMX Transitions 34-74
34.5.2.1 System-Wide Tracing 34-75
34.5.2.2 Guest-Only Tracing 34-75
34.5.2.3 Emulation of Intel PT Traced State 34-76
34.5.2.4 TSC Scaling 34-76
34.5.2.5 Failed VM Entry 34-76
34.5.2.6 VMX Abort 34-77
34.6 TRACING AND SMM TRANSFER MONITOR (STM) 34-77
34.7 PACKET GENERATION SCENARIOS 34-77
34.8 SOFTWARE CONSIDERATIONS 34-81
34.8.1 Tracing SMM Code 34-81
34.8.2 Cooperative Transition of Multiple Trace Collection Agents 34-81
34.8.3 Tracking Time 34-81
34.8.3.1 Time Domain Relationships 34-82
34.8.3.2 Estimating TSC within Intel PT 34-82
34.8.3.3 VMX TSC Manipulation 34-83
34.8.3.4 Calculating Frequency with Intel PT 34-83

CHAPTER 35
INTRODUCTION TO INTEL® SOFTWARE GUARD EXTENSIONS

35.1 OVERVIEW 35-1
35.2 ENCLAVE INTERACTION AND PROTECTION 35-1
35.3 ENCLAVE LIFE CYCLE 35-2
35.4 DATA STRUCTURES AND ENCLAVE OPERATION 35-2
35.5 ENCLAVE PAGE CACHE 35-2
35.5.1 Enclave Page Cache Map (EPCM) 35-3
35.6 ENCLAVE INSTRUCTIONS AND INTEL® SGX 35-3
35.7 DISCOVERING SUPPORT FOR INTEL® SGX AND ENABLING ENCLAVE INSTRUCTIONS 35-4
35.7.1 Intel® SGX Opt-In Configuration 35-5
35.7.2 Intel® SGX Resource Enumeration Leaves 35-5
35.8 INTEL® SGX INTERACTIONS WITH CONTROL-FLOW ENFORCEMENT TECHNOLOGY 35-7
35.8.1 CET in Enclaves Model 35-7
35.8.2 Operations Not Supported on Shadow Stack Pages 35-8
35.8.3 Indirect Branch Tracking – Legacy Compatibility Treatment 35-8

CHAPTER 36
ENCLAVE ACCESS CONTROL AND DATA STRUCTURES

36.1 OVERVIEW OF ENCLAVE EXECUTION ENVIRONMENT 36-1
36.2 TERMINOLOGY 36-1
36.3 ACCESS-CONTROL REQUIREMENTS 36-1
36.4 SEGMENT-BASED ACCESS CONTROL 36-2
36.5 PAGE-BASED ACCESS CONTROL 36-2
36.5.1 Access-control for Accesses that Originate from Non-SGX Instructions 36-2
36.5.2 Memory Accesses that Split Across ELRANGE 36-2
36.5.3 Implicit vs. Explicit Accesses 36-3
36.5.3.1 Explicit Accesses 36-3
36.5.3.2 Implicit Accesses 36-3
36.6 INTEL® SGX DATA STRUCTURES OVERVIEW 36-4
36.7 SGX ENCLAVE CONTROL STRUCTURE (SECS) 36-5
36.7.1 ATTRIBUTES 36-6
36.7.2 SECS.MISCSELECT Field 36-6
36.7.3 SECS.CET_ATTRIBUTES Field 36-6
36.8 THREAD CONTROL STRUCTURE (TCS) 36-7
36.8.1 TCS.FLAGS 36-8
36.8.2 State Save Area Offset (OSSA) 36-8
36.8.3 Current State Save Area Frame (CSSA) 36-8
36.8.4 Number of State Save Area Frames (NSSA) 36-8
36.9 STATE SAVE AREA (SSA) FRAME 36-8
36.9.1 GPRSGX Region 36-9
36.9.1.1 EXITINFO 36-10
36.9.1.2 VECTOR Field Definition 36-10
36.9.2 MISC Region 36-11

PAGE

36.9.2.1 EXINFO Structure....36-11
36.9.2.2 Page Fault Error Code....36-12
36.10 CET STATE SAVE AREA FRAME....36-12
36.11 PAGE INFORMATION (PAGEINFO)....36-12
36.12 SECURITY INFORMATION (SECINFO)....36-12
36.12.1 SECINFO.FLAGS....36-13
36.12.2 PAGE_TYPE Field Definition....36-13
36.13 PAGING CRYPTO METADATA (PCMD)....36-13
36.14 ENCLAVE SIGNATURE STRUCTURE (SIGSTRUCT)....36-14
36.15 EINIT TOKEN STRUCTURE (EINITTOKEN)....36-15
36.16 REPORT (REPORT)....36-16
36.16.1 REPORTDATA....36-17
36.17 REPORT TARGET INFO (TARGETINFO)....36-17
36.18 KEY REQUEST (KEYREQUEST)....36-17
36.18.1 KEY REQUEST KeyNames....36-18
36.18.2 Key Request Policy Structure....36-18
36.19 VERSION ARRAY (VA)....36-19
36.20 ENCLAVE PAGE CACHE MAP (EPCM)....36-19
36.21 READ INFO (RDINFO)....36-19
36.21.1 RDINFO Status Structure....36-20
36.21.2 RDINFO Flags Structure....36-20

CHAPTER 37
ENCLAVE OPERATION

37.1 CONSTRUCTING AN ENCLAVE....37-1
37.1.1 ECREATE....37-2
37.1.2 EADD and EEXTEND Interaction....37-2
37.1.3 EINIT Interaction....37-2
37.1.4 Intel® SGX Launch Control Configuration....37-3
37.2 ENCLAVE ENTRY AND EXITING....37-3
37.2.1 Controlled Entry and Exit....37-3
37.2.2 Asynchronous Enclave Exit (AEX)....37-4
37.2.3 Resuming Execution After AEX....37-4
37.2.3.1 ERESUME Interaction....37-5
37.2.3.2 Asynchronous Enclave Exit Notify and EDECCSSA....37-5
37.3 CALLING ENCLAVE PROCEDURES....37-6
37.3.1 Calling Convention....37-6
37.3.2 Register Preservation....37-6
37.3.3 Returning to Caller....37-6
37.4 INTEL® SGX KEY AND ATTESTATION....37-6
37.4.1 Enclave Measurement and Identification....37-6
37.4.1.1 MRENCLAVE....37-6
37.4.1.2 MRSIGNER....37-7
37.4.1.3 CONFIGID....37-7
37.4.2 Security Version Numbers (SVN)....37-7
37.4.2.1 Enclave Security Version....37-8
37.4.2.2 Hardware Security Version....37-8
37.4.2.3 CONFIGID Security Version....37-8
37.4.3 Keys....37-8
37.4.3.1 Sealing Enclave Data....37-9
37.4.3.2 Using REPORTs for Local Attestation....37-9
37.5 EPC AND MANAGEMENT OF EPC PAGES....37-10
37.5.1 EPC Implementation....37-10
37.5.2 OS Management of EPC Pages....37-10
37.5.2.1 Enhancement to Managing EPC Pages....37-10
37.5.3 Eviction of Enclave Pages....37-10
37.5.4 Loading an Enclave Page....37-11
37.5.5 Eviction of an SECS Page....37-11
37.5.6 Eviction of a Version Array Page....37-12
37.5.7 Allocating a Regular Page....37-12
37.5.8 Allocating a TCS Page....37-12
37.5.9 Trimming a Page....37-13
37.5.10 Restricting the EPCM Permissions of a Page....37-13
37.5.11 Extending the EPCM Permissions of a Page....37-14

37.5.12 VMM Oversubscription of EPC 37-14
37.6 CHANGES TO INSTRUCTION BEHAVIOR INSIDE AN ENCLAVE 37-14
37.6.1 Illegal Instructions 37-15
37.6.2 RDRAND and RDSEED Instructions 37-15
37.6.3 PAUSE Instruction 37-15
37.6.4 Executions of INT1 and INT3 Inside an Enclave 37-16
37.6.5 INVD Handling when Enclaves Are Enabled 37-16

CHAPTER 38
ENCLAVE EXITING EVENTS

38.1 COMPATIBLE SWITCH TO THE EXITING STACK OF AEX 38-1
38.2 STATE SAVING BY AEX 38-2
38.3 SYNTHETIC STATE ON ASYNCHRONOUS ENCLAVE EXIT 38-3
38.3.1 Processor Synthetic State on Asynchronous Enclave Exit 38-3
38.3.2 Synthetic State for Extended Features 38-3
38.3.3 Synthetic State for MISC Features 38-4
38.4 AEX FLOW 38-4
38.4.1 AEX Operational Detail 38-5

CHAPTER 39
INTEL® SGX INSTRUCTION REFERENCES

39.1 INTEL® SGX INSTRUCTION SYNTAX AND OPERATION 39-1
39.1.1 ENCLS Register Usage Summary 39-1
39.1.2 ENCLU Register Usage Summary 39-2
39.1.3 ENCLV Register Usage Summary 39-2
39.1.4 Information and Error Codes 39-2
39.1.5 Internal CREGs 39-3
39.1.6 Concurrent Operation Restrictions 39-4
39.1.6.1 Concurrency Tables of Intel® SGX Instructions 39-4
39.2 INTEL® SGX INSTRUCTION REFERENCE 39-8
ENCLS—Execute an Enclave System Function of Specified Leaf Number 39-9
ENCLU—Execute an Enclave User Function of Specified Leaf Number 39-11
ENCLV—Execute an Enclave VMM Function of Specified Leaf Number 39-14
39.3 INTEL® SGX SYSTEM LEAF FUNCTION REFERENCE 39-16
EADD—Add a Page to an Uninitialized Enclave 39-17
EAUG—Add a Page to an Initialized Enclave 39-22
EBLOCK—Mark a page in EPC as Blocked 39-27
ECREATE—Create an SECS page in the Enclave Page Cache 39-30
EDBGRD—Read From a Debug Enclave 39-36
EDBGWR—Write to a Debug Enclave 39-40
EEXTEND—Extend Uninitialized Enclave Measurement by 256 Bytes 39-44
EINIT—Initialize an Enclave for Execution 39-47
ELDB/ELDU/ELDBC/ELDUC—Load an EPC Page and Mark its State 39-55
EMODPR—Restrict the Permissions of an EPC Page 39-61
EMODT—Change the Type of an EPC Page 39-64
EPA—Add Version Array 39-67
ERDINFO—Read Type and Status Information About an EPC Page 39-69
EREMOVE—Remove a page from the EPC 39-73
ETRACK—Activates EBLOCK Checks 39-77
ETRACKC—Activates EBLOCK Checks 39-80
EUPDATESVN—Update CR_CPUSVN 39-84
EWB—Invalidate an EPC Page and Write out to Main Memory 39-86
39.4 INTEL® SGX USER LEAF FUNCTION REFERENCE 39-91
EACCEPT—Accept Changes to an EPC Page 39-92
EACCEPTCOPY—Initialize a Pending Page 39-97
EDECCSSA—Decrements TCS.CSSA 39-101
EENTER—Enters an Enclave 39-105
EEXIT—Exits an Enclave 39-114
EGETKEY—Retrieves a Cryptographic Key 39-117
EMODPE—Extend an EPC Page Permissions 39-127

PAGE

EREPORT—Create a Cryptographic Report of the Enclave 39-130
ERESUME—Re-Enters an Enclave 39-135
39.5 INTEL® SGX VIRTUALIZATION LEAF FUNCTION REFERENCE 39-147
EDECVIRTCHILD—Decrement VIRTCHILDCNT in SECS 39-148
EINCVIRTCHILD—Increment VIRTCHILDCNT in SECS 39-152
ESETCONTEXT—Set the ENCLAVECONTEXT Field in SECS 39-155

CHAPTER 40
INTEL® SGX INTERACTIONS WITH IA32 AND INTEL® 64 ARCHITECTURE
40.1 INTEL® SGX AVAILABILITY IN VARIOUS PROCESSOR MODES 40-1
40.2 IA32_FEATURE_CONTROL 40-1
40.2.1 Availability of Intel SGX 40-1
40.2.2 Intel SGX Launch Control Configuration 40-1
40.3 INTERACTIONS WITH SEGMENTATION 40-1
40.3.1 Scope of Interaction 40-1
40.3.2 Interactions of Intel® SGX Instructions with Segment, Operand, and Addressing Prefixes 40-2
40.3.3 Interaction of Intel® SGX Instructions with Segmentation 40-2
40.3.4 Interactions of Enclave Execution with Segmentation 40-2
40.4 INTERACTIONS WITH PAGING 40-2
40.5 INTERACTIONS WITH VMX 40-3
40.5.1 VMM Controls to Configure Guest Support of Intel® SGX 40-3
40.5.2 Interactions with the Extended Page Table Mechanism (EPT) 40-3
40.5.3 Interactions with APIC Virtualization 40-4
40.5.4 Interactions with VT and SGX concurrency 40-4
40.5.5 Virtual Child Tracking 40-5
40.5.6 Handling EPCM Entry Lock Conflicts 40-5
40.5.7 Context Tracking 40-6
40.6 INTEL® SGX INTERACTIONS WITH ARCHITECTURALLY-VISIBLE EVENTS 40-6
40.7 INTERACTIONS WITH THE PROCESSOR EXTENDED STATE AND MISCELLANEOUS STATE 40-6
40.7.1 Requirements and Architecture Overview 40-6
40.7.2 Relevant Fields in Various Data Structures 40-7
40.7.2.1 SECS.ATTRIBUTES.XFRM 40-7
40.7.2.2 SECS.SSAFRAMESIZE 40-8
40.7.2.3 XSAVE Area in SSA 40-8
40.7.2.4 MISC Area in SSA 40-8
40.7.2.5 SIGSTRUCT Fields 40-8
40.7.2.6 REPORT.ATTRIBUTES.XFRM and REPORT.MISCSELECT 40-9
40.7.2.7 KEYREQUEST 40-9
40.7.3 Processor Extended States and ENCLS[ECREATE] 40-9
40.7.4 Processor Extended States and ENCLU[EENTER] 40-9
40.7.4.1 Fault Checking 40-9
40.7.4.2 State Loading 40-9
40.7.5 Processor Extended States and AEX 40-10
40.7.5.1 State Saving 40-10
40.7.5.2 State Synthesis 40-10
40.7.6 Processor Extended States and ENCLU[ERESUME] 40-10
40.7.6.1 Fault Checking 40-10
40.7.6.2 State Loading 40-10
40.7.7 Processor Extended States and ENCLU[EEXIT] 40-10
40.7.8 Processor Extended States and ENCLU[EREPORT] 40-11
40.7.9 Processor Extended States and ENCLU[EGETKEY] 40-11
40.8 INTERACTIONS WITH SMM 40-11
40.8.1 Availability of Intel® SGX instructions in SMM 40-11
40.8.2 SMI while Inside an Enclave 40-11
40.8.3 SMRAM Synthetic State of AEX Triggered by SMI 40-11
40.9 INTERACTIONS OF INIT, SIPI, AND WAIT-FOR-SIPI WITH INTEL® SGX 40-12
40.10 INTERACTIONS WITH DMA 40-12
40.11 INTERACTIONS WITH TXT 40-12
40.11.1 Enclaves Created Prior to Execution of GETSEC 40-12
40.11.2 Interaction of GETSEC with Intel® SGX 40-12
40.11.3 Interactions with Authenticated Code Modules (ACMs) 40-13
40.12 INTERACTIONS WITH CACHING OF LINEAR-ADDRESS TRANSLATIONS 40-13
40.13 INTERACTIONS WITH INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS (INTEL® TSX) 40-13

40.13.1 HLE and RTM Debug 40-14
40.14 INTEL® SGX INTERACTIONS WITH S STATES 40-14
40.15 INTEL® SGX INTERACTIONS WITH MACHINE CHECK ARCHITECTURE (MCA) 40-14
40.15.1 Interactions with MCA Events 40-14
40.15.2 Machine Check Enables (IA32_MCi_CTL) 40-14
40.15.3 CR4.MCE 40-14
40.16 INTEL® SGX INTERACTIONS WITH PROTECTED MODE VIRTUAL INTERRUPTS 40-15
40.17 INTEL SGX INTERACTION WITH PROTECTION KEYS 40-15

CHAPTER 41
ENCLAVE CODE DEBUG AND PROFILING

41.1 CONFIGURATION AND CONTROLS 41-1
41.1.1 Debug Enclave vs. Production Enclave 41-1
41.1.2 Tool-Chain Opt-in 41-1
41.1.3 Debugging an Enclave That Uses Asynchronous Enclave Exit Notify 41-1
41.2 SINGLE STEP DEBUG 41-1
41.2.1 Single Stepping ENCLS Instruction Leafs 41-1
41.2.2 Single Stepping ENCLU Instruction Leafs 41-2
41.2.3 Single-Stepping Enclave Entry with Opt-out Entry 41-2
41.2.3.1 Single Stepping without AEX 41-2
41.2.3.2 Single Step Preempted by AEX Due to Non-SMI Event 41-2
41.2.4 RFLAGS.TF Treatment on AEX 41-3
41.2.5 Restriction on Setting of TF after an Opt-Out Entry 41-3
41.2.6 Trampoline Code Considerations 41-3
41.3 CODE AND DATA BREAKPOINTS 41-3
41.3.1 Breakpoint Suppression 41-3
41.3.2 Reporting of Instruction Breakpoint on Next Instruction on a Debug Trap 41-4
41.3.3 RF Treatment on AEX 41-4
41.3.4 Breakpoint Matching in Intel® SGX Instruction Flows 41-4
41.4 CONSIDERATION OF THE INT1 AND INT3 INSTRUCTIONS 41-4
41.4.1 Behavior of INT1 and INT3 Inside an Enclave 41-4
41.4.2 Debugger Considerations 41-4
41.4.3 VMM Considerations 41-5
41.5 BRANCH TRACING 41-5
41.5.1 BTF Treatment 41-5
41.5.2 LBR Treatment 41-5
41.5.2.1 LBR Stack on Opt-in Entry 41-5
41.5.2.2 LBR Stack on Opt-out Entry 41-6
41.5.2.3 Mispredict Bit, Record Type, and Filtering 41-7
41.6 INTERACTION WITH PERFORMANCE MONITORING 41-7
41.6.1 IA32_PERF_GLOBAL_STATUS Enhancement 41-7
41.6.2 Performance Monitoring with Opt-in Entry 41-7
41.6.3 Performance Monitoring with Opt-out Entry 41-8
41.6.4 Enclave Exit and Performance Monitoring 41-8
41.6.5 PEBS Record Generation on Intel® SGX Instructions 41-8
41.6.6 Exception-Handling on PEBS/BTS Loads/Stores after AEX 41-8
41.6.6.1 Other Interactions with Performance Monitoring 41-9

APPENDIX A
VMX CAPABILITY REPORTING FACILITY

A.1 BASIC VMX INFORMATION A-1
A.2 RESERVED CONTROLS AND DEFAULT SETTINGS A-2
A.3 VM-EXECUTION CONTROLS A-2
A.3.1 Pin-Based VM-Execution Controls A-2
A.3.2 Primary Processor-Based VM-Execution Controls A-3
A.3.3 Secondary Processor-Based VM-Execution Controls A-4
A.3.4 Tertiary Processor-Based VM-Execution Controls A-4
A.4 VM-EXIT CONTROLS A-4
A.4.1 Primary VM-Exit Controls A-4
A.4.2 Secondary VM-Exit Controls A-5
A.5 VM-ENTRY CONTROLS A-5
A.6 MISCELLANEOUS DATA A-6

PAGE

A.7 VMX-FIXED BITS IN CR0 ..... A-7
A.8 VMX-FIXED BITS IN CR4 ..... A-7
A.9 VMCS ENUMERATION A-7
A.10 VPID AND EPT CAPABILITIES A-8
A.11 VM FUNCTIONS A-9

APPENDIX B
FIELD ENCODING IN VMCS

B.1 16-BIT FIELDS B-1
B.1.1 16-Bit Control Fields B-1
B.1.2 16-Bit Guest-State Fields B-1
B.1.3 16-Bit Host-State Fields B-2
B.2 64-BIT FIELDS B-2
B.2.1 64-Bit Control Fields B-2
B.2.2 64-Bit Read-Only Data Fields B-5
B.2.3 64-Bit Guest-State Fields B-5
B.2.4 64-Bit Host-State Fields B-7
B.3 32-BIT FIELDS B-7
B.3.1 32-Bit Control Fields B-7
B.3.2 32-Bit Read-Only Data Fields B-8
B.3.3 32-Bit Guest-State Fields B-8
B.3.4 32-Bit Host-State Field B-9
B.4 NATURAL-WIDTH FIELDS B-9
B.4.1 Natural-Width Control Fields B-9
B.4.2 Natural-Width Read-Only Data Fields B-10
B.4.3 Natural-Width Guest-State Fields B-10
B.4.4 Natural-Width Host-State Fields B-11

APPENDIX C
VMX BASIC EXIT REASONS

FIGURES

Figure 2-1. IA-32 System-Level Registers and Data Structures....2-2
Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode and 4-Level Paging....2-3
Figure 2-3. Transitions Among the Processor's Operating Modes....2-8
Figure 2-4. IA32_EFER MSR Layout....2-9
Figure 2-5. System Flags in the EFLAGS Register....2-10
Figure 2-6. Memory Management Registers....2-12
Figure 2-7. Control Registers....2-14
Figure 2-8. XCR0....2-21
Figure 2-9. Format of Protection-Key Rights Registers....2-22
Figure 2-10. WBINVD Invalidation of Shared and Non-Shared Cache Hierarchy....2-26
Figure 3-1. Segmentation and Paging....3-2
Figure 3-2. Flat Model....3-3
Figure 3-3. Protected Flat Model....3-4
Figure 3-4. Multi-Segment Model....3-5
Figure 3-5. Logical Address to Linear Address Translation....3-7
Figure 3-6. Segment Selector....3-7
Figure 3-7. Segment Registers....3-8
Figure 3-8. Segment Descriptor....3-10
Figure 3-9. Segment Descriptor When Segment-Present Flag Is Clear....3-11
Figure 3-10. Global and Local Descriptor Tables....3-15
Figure 3-11. Pseudo-Descriptor Formats....3-16
Figure 5-1. Enabling and Changing Paging Modes....5-4
Figure 5-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging....5-10
Figure 5-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging....5-11
Figure 5-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging....5-11
Figure 5-5. Linear-Address Translation to a 4-KByte Page using PAE Paging....5-16
Figure 5-6. Linear-Address Translation to a 2-MByte Page using PAE Paging....5-17
Figure 5-7. Formats of CR3 and Paging-Structure Entries with PAE Paging....5-19
Figure 5-8. Linear-Address Translation to a 4-KByte Page Using 4-Level Paging....5-22
Figure 5-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging....5-23
Figure 5-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging....5-23
Figure 5-11. Formats of CR3 and Paging-Structure Entries with 4-Level Paging and 5-Level Paging....5-32
Figure 5-12. Page-Fault Error Code....5-37
Figure 5-13. Memory Management Convention That Assigns a Page Table to Each Segment....5-55
Figure 6-1. Descriptor Fields Used for Protection....6-3
Figure 6-2. Descriptor Fields with Flags used in IA-32e Mode....6-4
Figure 6-3. Protection Rings....6-7
Figure 6-4. Privilege Check for Data Access....6-8
Figure 6-5. Examples of Accessing Data Segments From Various Privilege Levels....6-9
Figure 6-6. Privilege Check for Control Transfer Without Using a Gate....6-11
Figure 6-7. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels....6-12
Figure 6-8. Call-Gate Descriptor....6-13
Figure 6-9. Call-Gate Descriptor in IA-32e Mode....6-14
Figure 6-10. Call-Gate Mechanism....6-15
Figure 6-11. Privilege Check for Control Transfer with Call Gate....6-16
Figure 6-12. Example of Accessing Call Gates At Various Privilege Levels....6-17
Figure 6-13. Stack Switching During an Interprivilege-Level Call....6-19
Figure 6-14. MSRs Used by SYSCALL and SYSRET....6-23
Figure 6-15. Use of RPL to Weaken Privilege Level of Called Procedure....6-26
Figure 7-1. Relationship of the IDTR and IDT....7-10
Figure 7-2. IDT Gate Descriptors....7-11
Figure 7-3. Interrupt Procedure Call....7-12
Figure 7-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines....7-13
Figure 7-5. Shadow Stack Usage on Transfers to Interrupt and Exception-Handling Routines....7-15
Figure 7-6. Interrupt Task Switch....7-18
Figure 7-7. Error Code....7-19
Figure 7-8. 64-Bit IDT Gate Descriptors....7-20
Figure 7-9. IA-32e Mode Stack Usage After Privilege Level Change....7-22
Figure 7-10. Interrupt Shadow Stack Table....7-23
Figure 7-11. Page-Fault Error Code....7-46
Figure 7-12. Exception Error Code Information....7-56
Figure 9-1. Structure of a Task....9-2

PAGE

Figure 9-2. 32-Bit Task-State Segment (TSS) 9-4
Figure 9-3. TSS Descriptor 9-6
Figure 9-4. Format of TSS and LDT Descriptors in 64-bit Mode 9-7
Figure 9-5. Task Register 9-8
Figure 9-6. Task-Gate Descriptor 9-8
Figure 9-7. Task Gates Referencing the Same Task 9-9
Figure 9-8. Nested Tasks 9-15
Figure 9-9. Overlapping Linear-to-Physical Mappings 9-17
Figure 9-10. 16-Bit TSS Format 9-19
Figure 9-11. 64-Bit TSS Format 9-20
Figure 10-1. Example of Write Ordering in Multiple-Processor Systems 10-8
Figure 10-2. Interpretation of APIC ID in Early MP Systems 10-26
Figure 10-3. Local APICs and I/O APIC in MP System Supporting Intel HT Technology 10-28
Figure 10-4. IA-32 Processor with Two Logical Processors Supporting Intel HT Technology 10-29
Figure 10-5. Generalized Seven-Domain Interpretation of the APIC ID 10-36
Figure 10-6. Conceptual Six-Domain Topology and 32-bit APIC ID Composition 10-37
Figure 10-7. Topological Relationships Between Hierarchical IDs in a Hypothetical MP Platform 10-40
Figure 10-8. MP System With Multiple Pentium III Processors 10-57
Figure 11-1. Contents of CR0 Register after Reset 11-2
Figure 11-2. Version Information in the EDX Register after Reset 11-5
Figure 11-3. Processor State After Reset 11-15
Figure 11-4. Constructing Temporary GDT and Switching to Protected Mode (Lines 162-172 of List File) 11-23
Figure 11-5. Moving the GDT, IDT, and TSS from ROM to RAM (Lines 196-261 of List File) 11-24
Figure 11-6. Task Switching (Lines 282-296 of List File) 11-25
Figure 11-7. Applying Microcode Updates 11-28
Figure 11-8. Microcode Update Write Operation Flow [1] 11-45
Figure 11-9. Microcode Update Write Operation Flow [2] 11-46
Figure 12-1. Relationship of Local APIC and I/O APIC In Single-Processor Systems 12-2
Figure 12-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems 12-3
Figure 12-3. Local APICs and I/O APIC When P6 Family Processors Are Used in Multiple-Processor Systems 12-3
Figure 12-4. Local APIC Structure 12-5
Figure 12-5. IA32_APIC_BASE MSR (APIC_BASE_MSR in P6 Family) 12-9
Figure 12-6. Local APIC ID Register 12-9
Figure 12-7. Local APIC Version Register 12-11
Figure 12-8. Local Vector Table (LVT) 12-13
Figure 12-9. Error Status Register (ESR) 12-15
Figure 12-10. Divide Configuration Register 12-17
Figure 12-11. Initial Count and Current Count Registers 12-17
Figure 12-12. Interrupt Command Register (ICR) 12-19
Figure 12-13. Logical Destination Register (LDR) 12-24
Figure 12-14. Destination Format Register (DFR) 12-24
Figure 12-15. Arbitration Priority Register (APR) 12-25
Figure 12-16. Interrupt Acceptance Flow Chart for the Local APIC (Pentium 4 and Intel Xeon Processors) 12-27
Figure 12-17. Interrupt Acceptance Flow Chart for the Local APIC (P6 Family and Pentium Processors) 12-28
Figure 12-18. Task-Priority Register (TPR) 12-29
Figure 12-19. Processor-Priority Register (PPR) 12-29
Figure 12-20. IRR, ISR, and TMR Registers 12-30
Figure 12-21. EOI Register 12-31
Figure 12-22. CR8 Register 12-32
Figure 12-23. Spurious-Interrupt Vector Register (SVR) 12-33
Figure 12-24. Layout of the MSI Message Address Register 12-34
Figure 12-25. Layout of the MSI Message Data Register 12-36
Figure 12-26. IA32_APIC_BASE MSR Supporting x2APIC 12-37
Figure 12-27. Local x2APIC State Transitions with IA32_APIC_BASE, INIT, and Reset 12-42
Figure 12-28. Interrupt Command Register (ICR) in x2APIC Mode 12-45
Figure 12-29. Logical Destination Register in x2APIC Mode 12-46
Figure 12-30. SELF IPI register 12-47
Figure 13-1. Cache Structure of the Pentium 4 and Intel Xeon Processors 13-1
Figure 13-2. Cache Structure of the Intel Core i7 Processors 13-2
Figure 13-3. Cache-Control Registers and Bits Available in Intel 64 and IA-32 Processors 13-11
Figure 13-4. Mapping Physical Memory With MTRRs 13-21
Figure 13-5. IA32_MTRRCAP Register 13-22
Figure 13-6. IA32_MTRR_DEF_TYPE MSR 13-23
Figure 13-7. IA32_MTRR_PHYSBASEn and IA32_MTRR_PHYSMASKn Variable-Range Register Pair 13-25
Figure 13-8. IA32_SMRR_PHYSBASE and IA32_SMRR_PHYSMASK SMRR Pair 13-26

PAGE

Figure 13-9. IA32_PAT MSR 13-34
Figure 14-1. Mapping of MMX Registers to Floating-Point Registers 14-2
Figure 14-2. Mapping of MMX Registers to x87 FPU Data Register Stack 14-5
Figure 16-1. IA32_MPERF MSR and IA32_APERF MSR for P-state Coordination 16-2
Figure 16-2. IA32_PERF_CTL Register 16-4
Figure 16-3. IA32_ENERGY_PERF_BIAS Register 16-5
Figure 16-4. IA32_PM_ENABLE MSR 16-7
Figure 16-5. IA32_HWP_CAPABILITIES Register 16-8
Figure 16-6. IA32_HWP_REQUEST Register 16-9
Figure 16-7. IA32_HWP_REQUEST_PKG Register 16-11
Figure 16-8. IA32_HWP_PECI_REQUEST_INFO MSR 16-11
Figure 16-9. IA32_HWP_STATUS MSR 16-14
Figure 16-10. IA32_THERM_STATUS Register With HWP Feedback 16-15
Figure 16-11. MSR_PPERF MSR 16-15
Figure 16-12. IA32_HWP_INTERRUPT MSR 16-16
Figure 16-13. FAST_UNCORE_MSRS_CAPABILITY MSR 16-17
Figure 16-14. FAST_UNCORE_MSRS_CTL MSR 16-18
Figure 16-15. FAST_UNCORE_MSRS_STATUS MSR 16-18
Figure 16-16. IA32_PKG_HDC_CTL MSR 16-21
Figure 16-17. IA32_PM_CTL1 MSR 16-22
Figure 16-18. IA32_THREAD_STALL MSR 16-22
Figure 16-19. MSR_CORE_HDC_RESIDENCY MSR 16-23
Figure 16-20. MSR_PKG_HDC_SHALLOW_RESIDENCY MSR 16-23
Figure 16-21. MSR_PKG_HDC_DEEP_RESIDENCY MSR 16-24
Figure 16-22. MSR_PKG_HDC_CONFIG MSR 16-24
Figure 16-23. Example of Effective Frequency Reduction and Forced Idle Period of HDC 16-25
Figure 16-24. Processor Modulation Through Stop-Clock Mechanism 16-36
Figure 16-25. MSR_THERM2_CTL Register On Processors with CPUID Family/Model/Stepping Signature Encoded as 0x69n or 0x6Dn 16-38
Figure 16-26. MSR_THERM2_CTL Register for Supporting TM2 16-38
Figure 16-27. IA32_THERM_STATUS MSR 16-39
Figure 16-28. IA32_THERM_INTERRUPT MSR 16-39
Figure 16-29. IA32_CLOCK_MODULATION MSR 16-40
Figure 16-30. IA32_CLOCK_MODULATION MSR with Clock Modulation Extension 16-41
Figure 16-31. IA32_THERM_STATUS Register 16-42
Figure 16-32. IA32_THERM_INTERRUPT Register 16-44
Figure 16-33. IA32_PACKAGE_THERM_STATUS Register 16-45
Figure 16-34. IA32_PACKAGE_THERM_INTERRUPT Register 16-47
Figure 16-35. MSR_RAPL_POWER_UNIT Register 16-49
Figure 16-36. MSR_PKG_POWER_LIMIT Register 16-50
Figure 16-37. MSR_PKG_ENERGY_STATUS MSR 16-51
Figure 16-38. MSR_PKG_POWER_INFO Register 16-51
Figure 16-39. MSR_PKG_PERF_STATUS MSR 16-52
Figure 16-40. MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT Register 16-52
Figure 16-41. MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS MSR 16-53
Figure 16-42. MSR_PP0_POLICY/MSR_PP1_POLICY Register 16-53
Figure 16-43. MSR_PP0_PERF_STATUS MSR 16-54
Figure 16-44. MSR_DRAM_POWER_LIMIT Register 16-54
Figure 16-45. MSR_DRAM_ENERGY_STATUS MSR 16-55
Figure 16-46. MSR_DRAM_POWER_INFO Register 16-55
Figure 16-47. MSR_DRAM_PERF_STATUS MSR 16-55
Figure 17-1. Machine-Check MSRs 17-2
Figure 17-2. IA32_MCG_CAP Register 17-3
Figure 17-3. IA32_MCG_STATUS Register 17-4
Figure 17-4. IA32_MCG_EXT_CTL Register 17-5
Figure 17-5. IA32_MCi_CTL Register 17-6
Figure 17-6. IA32_MCi_STATUS Register 17-7
Figure 17-7. IA32_MCi_ADDR MSR 17-9
Figure 17-8. UCR Support in IA32_MCi_MISC Register 17-10
Figure 17-9. IA32_MCi_CTL2 Register 17-11
Figure 17-10. CMCI Behavior 17-14
Figure 19-1. Debug Registers 19-3
Figure 19-2. DR6/DR7 Layout on Processors Supporting Intel® 64 Architecture 19-8
Figure 19-3. IA32_DEBUGCTL MSR for Processors Based on Intel® Core™ Microarchitecture 19-13
Figure 19-4. 64-bit Address Layout of LBR MSR 19-18

PAGE

Figure 19-5. DS Save Area Example 19-21
Figure 19-6. 32-bit Branch Trace Record Format 19-22
Figure 19-7. PEBS Record Format 19-22
Figure 19-8. IA-32e Mode DS Save Area Example 19-23
Figure 19-9. 64-bit Branch Trace Record Format 19-23
Figure 19-10. 64-bit PEBS Record Format 19-24
Figure 19-11. IA32_DEBUGCTL MSR for Processors Based on Nehalem Microarchitecture 19-30
Figure 19-12. MSR_DEBUGCTLA MSR for Pentium 4 and Intel Xeon Processors 19-36
Figure 19-13. LBR MSR Branch Record Layout for the Pentium 4 and Intel® Xeon® Processor Family 19-37
Figure 19-14. IA32_DEBUGCTL MSR for Intel® Core™ Solo and Intel® Core™ Duo Processors 19-38
Figure 19-15. LBR Branch Record Layout for the Intel® Core™ Solo and Intel® Core™ Duo Processor 19-39
Figure 19-16. MSR_DEBUGCTLB MSR for Pentium M Processors 19-40
Figure 19-17. LBR Branch Record Layout for the Pentium M Processor 19-40
Figure 19-18. DEBUGCTLMSR Register (P6 Family Processors) 19-41
Figure 19-19. Platform Shared Resource Monitoring Usage Flow 19-46
Figure 19-20. CPUID.(EAX=0FH, ECX=0H) Monitoring Resource Type Enumeration 19-47
Figure 19-21. L3 Cache Monitoring Capability Enumeration Data (CPUID.(EAX=0FH, ECX=1H)) 19-47
Figure 19-22. L3 Cache Monitoring Capability Enumeration Event Type Bit Vector (CPUID.(EAX=0FH, ECX=1H)) 19-48
Figure 19-23. IA32_PQR_ASSOC MSR 19-49
Figure 19-24. IA32_QM_EVTSEL and IA32_QM_CTR MSRs 19-51
Figure 19-25. Software Usage of Cache Monitoring Resources 19-51
Figure 19-26. Cache Allocation Technology Enables Allocation of More Resources to High Priority Applications 19-53
Figure 19-27. Examples of Cache Capacity Bitmasks 19-54
Figure 19-28. Class of Service and Cache Capacity Bitmasks 19-55
Figure 19-29. Code and Data Capacity Bitmasks of CDP 19-57
Figure 19-30. Cache Allocation Technology Usage Flow 19-57
Figure 19-31. CPUID.(EAX=10H, ECX=0H) Available Resource Type Identification 19-58
Figure 19-32. L3 Cache Allocation Technology and CDP Enumeration 19-59
Figure 19-33. L2 Cache Allocation Technology 19-60
Figure 19-34. IA32_PQR_ASSOC, IA32_L3_MASK_n MSRs 19-61
Figure 19-35. IA32_L2_MASK_n MSRs 19-61
Figure 19-36. Layout of IA32_L3_QOS_CFG 19-62
Figure 19-37. Layout of IA32_L2_QOS_CFG 19-64
Figure 19-38. CPUID.(EAX=10H, ECX=3H) MBA Feature Details Identification 19-67
Figure 19-39. IA32_L2_QoS_Ext_BW_Thrtl_n MSR Definition 19-68
Figure 19-40. Layout of the IA32_L3_IO_RDT_CFG MSR for Enabling Non-CPU Agent Intel® RDT 19-71
Figure 21-1. Layout of IA32_PERFEVTSELx MSRs 21-4
Figure 21-2. Layout of IA32_FIXED_CTR_CTRL MSR 21-6
Figure 21-3. Layout of IA32_PERF_GLOBAL_CTRL MSR 21-7
Figure 21-4. Layout of IA32_PERF_GLOBAL_STATUS MSR 21-8
Figure 21-5. Layout of IA32_PERF_GLOBAL_OVF_CTRL MSR 21-9
Figure 21-6. Layout of IA32_PERFEVTSELx MSRs Supporting Architectural Performance Monitoring Version 3 21-9
Figure 21-7. IA32_FIXED_CTR_CTRL MSR Supporting Architectural Performance Monitoring Version 3 21-10
Figure 21-8. Layout of Global Performance Monitoring Control MSR 21-10
Figure 21-9. Global Performance Monitoring Overflow Status and Control MSRs 21-11
Figure 21-10. IA32_PERF_GLOBAL_STATUS MSR and Architectural PerfMon Version 4 21-13
Figure 21-11. IA32_PERF_GLOBAL_STATUS_RESET MSR and Architectural PerfMon Version 4 21-13
Figure 21-12. IA32_PERF_GLOBAL_STATUS_SET MSR and Architectural PerfMon Version 4 21-14
Figure 21-13. IA32_PERF_GLOBAL_INUSE MSR and Architectural PerfMon Version 4 21-14
Figure 21-14. IA32_PMC_GPx_CFG_A MSR (also known as IA32_PERFEVTSELx) Supporting Architectural Performance Monitoring Version 6 21-17
Figure 21-15. IA32_FIXED_CTR_CTRL MSR Supporting Architectural Performance Monitoring Version 6 21-17
Figure 21-16. IA32_PERF_GLOBAL_STATUS MSR 21-23
Figure 21-17. Layout of IA32_PEBS_ENABLE MSR 21-25
Figure 21-18. PEBS Programming Environment 21-27
Figure 21-19. Layout of MSR_PEBS_LD_LAT MSR 21-29
Figure 21-20. Layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 to Configure Off-core Response Events 21-30
Figure 21-21. Layout of MSR_UNCORE_PERF_GLOBAL_CTRL MSR 21-32
Figure 21-22. Layout of MSR_UNCORE_PERF_GLOBAL_STATUS MSR 21-33
Figure 21-23. Layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL MSR 21-33
Figure 21-24. Layout of MSR_UNCORE_PERFEVTSELx MSRs 21-34
Figure 21-25. Layout of MSR_UNCORE_FIXED_CTR_CTRL MSR 21-34
Figure 21-26. Layout of MSR_UNCORE_ADDR_OPCODE_MATCH MSR 21-35
Figure 21-27. Distributed Units of the Uncore of Intel® Xeon® Processor 7500 Series 21-36
Figure 21-28. IA32_PERF_GLOBAL_CTRL MSR in Sandy Bridge Microarchitecture 21-39

PAGE

Figure 21-29. IA32_PERF_GLOBAL_STATUS MSR in Sandy Bridge Microarchitecture. 21-40
Figure 21-30. IA32_PERF_GLOBAL_OVF_CTRL MSR in Sandy Bridge Microarchitecture 21-41
Figure 21-31. Layout of IA32_PEBS_ENABLE MSR 21-43
Figure 21-32. Request_Type Fields for MSR_OFFCORE_RSP_x 21-47
Figure 21-33. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSP_x 21-48
Figure 21-34. Layout of Uncore PERFEVTSEL MSR for a C-Box Unit or the ARB Unit 21-49
Figure 21-35. Layout of MSR_UNC_PERF_GLOBAL_CTRL MSR for Uncore 21-50
Figure 21-36. Layout of IA32_PERFEVTSELx MSRs Supporting Intel TSX. 21-59
Figure 21-37. IA32_PERF_GLOBAL_STATUS MSR in Broadwell Microarchitecture. 21-61
Figure 21-38. IA32_PERF_GLOBAL_OVF_CTRL MSR in Broadwell microarchitecture 21-61
Figure 21-39. MSR_PERF_METRICS Definition 21-74
Figure 21-40. PERF_METRICS MSR Definition for 12th Generation Intel® Core™ Processor P-core 21-76
Figure 21-41. Deducing Implied Level 2 Metrics in the Core PMU for12th Generation Intel® Core™ Processor P-core. 21-77
Figure 21-42. Request_Type Fields for MSR_OFFCORE_RSPx 21-90
Figure 21-43. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSPx 21-91
Figure 21-44. IA32_PEBS_ENABLE MSR with PEBS Output to Intel® Processor Trace 21-100
Figure 21-45. Layout of IA32_FIXED_CTR_CTRL MSR 21-106
Figure 21-46. Layout of MSR_PERF_GLOBAL_CTRL MSR 21-107
Figure 21-47. Layout of MSR_PERF_GLOBAL_STATUS MSR 21-107
Figure 21-48. Layout of MSR_PERF_GLOBAL_OVF_CTRL MSR 21-108
Figure 21-49. Event Selection Control Register (ESCR) for Pentium 4 and Intel® Xeon® Processors without Intel HT Technology Support 21-114
Figure 21-50. Performance Counter (Pentium 4 and Intel® Xeon® Processors) 21-116
Figure 21-51. Counter Configuration Control Register (CCCR) 21-117
Figure 21-52. Effects of Edge Filtering 21-120
Figure 21-53. Event Selection Control Register (ESCR) for the Pentium 4 Processor, Intel® Xeon® Processor, and Intel® Xeon® Processor MP Supporting Hyper-Threading Technology 21-128
Figure 21-54. Counter Configuration Control Register (CCCR) 21-129
Figure 21-55. Block Diagram of 64-bit Intel® Xeon® Processor MP with 8-MByte L3 21-132
Figure 21-56. MSR_IFSB_IBUSQx, Addresses: 107CCH and 107CDH. 21-133
Figure 21-57. MSR_IFSB_ISNPQx, Addresses: 107CEH and 107CFH. 21-133
Figure 21-58. MSR_EFSB_DRDYx, Addresses: 107D0H and 107D1H 21-134
Figure 21-59. MSR_IFSB_CTL6, Address: 107D2H; MSR_IFSB_CNTR7, Address: 107D3H. 21-134
Figure 21-60. Block Diagram of the Intel® Xeon® Processor 7400 Series. 21-135
Figure 21-61. Block Diagram of the Intel® Xeon® Processor 7100 Series. 21-136
Figure 21-62. MSR_EMON_L3_CTR_CTL0/1, Addresses: 107CCH/107CDH 21-137
Figure 21-63. MSR_EMON_L3_CTR_CTL2/3, Addresses: 107CEH/107CFH. 21-139
Figure 21-64. MSR_EMON_L3_CTR_CTL4/5/6/7, Addresses: 107D0H-107D3H. 21-139
Figure 21-65. PerfEvtSel0 and PerfEvtSel1 MSRs. 21-141
Figure 21-66. CESR MSR (Pentium Processor Only). 21-144
Figure 21-67. Layout of IA32_PERF_CAPABILITIES MSR. 21-149
Figure 21-68. Layout of IA32_PEBS_ENABLE MSR 21-150
Figure 21-69. PEBS Programming Environment 21-151
Figure 21-70. Layout of IA32_PerfEvtSelX MSR Supporting Adaptive PEBS 21-152
Figure 21-71. Layout of IA32_FIXED_CTR_CTRL MSR Supporting Adaptive PEBS 21-153
Figure 21-72. Legacy MSR_PEBS_DATA_CFG 21-157
Figure 21-73. MSR_PEBS_DATA_CFG in PEBS_FMT=6 21-158
Figure 22-1. Real-Address Mode Address Translation 22-3
Figure 22-2. Interrupt Vector Table in Real-Address Mode 22-5
Figure 22-3. Entering and Leaving Virtual-8086 Mode 22-9
Figure 22-4. Privilege Level 0 Stack After Interrupt or Exception in Virtual-8086 Mode. 22-13
Figure 22-5. Software Interrupt Redirection Bit Map in TSS 22-18
Figure 23-1. Stack after Far 16- and 32-Bit Calls 23-5
Figure 24-1. I/O Map Base Address Differences. 24-30
Figure 25-1. Interaction of a Virtual-Machine Monitor and Guests. 25-2
Figure 26-1. States of VMCS X 26-2
Figure 30-1. Formats of EPTP and EPT Paging-Structure Entries 30-13
Figure 32-1. INVEPT Descriptor. 32-3
Figure 32-2. INVVPID Descriptor 32-6
Figure 33-1. SMRAM Usage. 33-4
Figure 33-2. SMM Revision Identifier 33-13
Figure 33-3. Auto HALT Restart Field 33-14
Figure 33-4. SMBASE Relocation Field. 33-15
Figure 33-5. I/O Instruction Restart Field. 33-15
Figure 34-1. ToPA Memory Illustration 34-11

PAGE

Figure 34-2. Layout of ToPA Table Entry 34-12
Figure 34-3. Interpreting Tabular Definition of Packet Format 34-38
Figure 35-1. An Enclave Within the Application's Virtual Address Space 35-1
Figure 37-1. Enclave Memory Layout 37-1
Figure 37-2. Measurement Flow of Enclave Build Process 37-7
Figure 37-3. SGX Local Attestation 37-9
Figure 38-1. Exit Stack Just After Interrupt with Stack Switch 38-1
Figure 38-2. The SSA Stack 38-2
Figure 39-1. Relationships Between SECS, SIGSTRUCT, and EINITTOKEN 39-47
Figure 41-1. Single Stepping with Opt-out Entry - No AEX 41-2
Figure 41-2. Single Stepping with Opt-out Entry -AEX Due to Non-SMI Event Before Single-Step Boundary 41-3
Figure 41-3. LBR Stack Interaction with Opt-in Entry 41-6
Figure 41-4. LBR Stack Interaction with Opt-out Entry 41-7

TABLES

Table 2-1. IA32_EFER MSR Information 2-9
Table 2-2. Action Taken By x87 FPU Instructions for Different Combinations of EM, MP, and TS 2-16
Table 2-3. Summary of System Instructions 2-23
Table 3-1. Code- and Data-Segment Types 3-12
Table 3-2. System-Segment and Gate-Descriptor Types 3-14
Table 5-1. Properties of Different Paging Modes 5-2
Table 5-2. Paging Structures in the Different Paging Modes 5-8
Table 5-3. Use of CR3 with 32-Bit Paging 5-12
Table 5-4. Format of a 32-Bit Page-Directory Entry that Maps a 4-MByte Page 5-12
Table 5-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page 5-13
Table 5-5. Format of a 32-Bit Page-Directory Entry that References a Page Table 5-13
Table 5-7. Use of CR3 with PAE Paging 5-14
Table 5-8. Format of a PAE Page-Directory-Pointer-Table Entry (PDPTE) 5-15
Table 5-9. Format of a PAE Page-Directory Entry that Maps a 2-MByte Page 5-17
Table 5-10. Format of a PAE Page-Directory Entry that References a Page Table 5-18
Table 5-11. Format of a PAE Page-Table Entry that Maps a 4-KByte Page 5-18
Table 5-12. Use of CR3 with 4-Level Paging and 5-level Paging and CR4.PCIDE = 0 5-20
Table 5-13. Use of CR3 with 4-Level Paging and 5-Level Paging and CR4.PCIDE = 1 5-21
Table 5-14. Format of a PML5 Entry (PML5E) that References a PML4 Table 5-26
Table 5-15. Format of a PML4 Entry (PML4E) that References a Page-Directory-Pointer Table 5-26
Table 5-16. Format of a Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page 5-27
Table 5-17. Format of a Page-Directory-Pointer-Table Entry (PDPTE) that References a Page Directory 5-28
Table 5-18. Format of a Page-Directory Entry that Maps a 2-MByte Page 5-29
Table 5-19. Format of a Page-Directory Entry that References a Page Table 5-30
Table 5-20. Format of a Page-Table Entry that Maps a 4-KByte Page 5-31
Table 6-1. Privilege Check Rules for Call Gates 6-16
Table 6-2. 64-Bit-Mode Stack Layout After Far CALL with CPL Change 6-19
Table 6-3. Combined Page-Directory and Page-Table Protection 6-29
Table 6-4. Extended Feature Enable MSR (IA32_EFER) 6-30
Table 6-6. 4-KByte Page Level Protection Matrix with Execute-Disable Bit Capability with PAE Paging 6-31
Table 6-7. 2-MByte Page Level Protection with Execute-Disable Bit Capability with PAE Paging 6-31
Table 6-5. Page Level Protection Matrix with Execute-Disable Bit Capability with 4-Level Paging 6-31
Table 6-9. Reserved Bit Checking with Execute-Disable Bit Capability Not Enabled 6-32
Table 6-8. Page Level Protection Matrix with Execute-Disable Bit Capability Enabled 6-32
Table 7-1. Protected-Mode Exceptions and Interrupts 7-2
Table 7-2. Priority Among Concurrent Events 7-8
Table 7-3. Debug Exception Conditions and Corresponding Exception Classes 7-25
Table 7-4. Interrupt and Exception Classes 7-33
Table 7-5. Conditions for Generating a Double Fault 7-34
Table 7-6. Invalid TSS Conditions 7-36
Table 7-7. Alignment Requirements by Data Type 7-50
Table 7-8. SIMD Floating-Point Exceptions Priority 7-54
Table 8-1. Format of User Posted-Interrupt Descriptor — UPID 8-5
Table 9-1. Exception Conditions Checked During a Task Switch 9-13
Table 9-2. Effect of a Task Switch on Busy Flag, NT Flag, Previous Task Link Field, and TS Flag 9-15
Table 10-1. Broadcast INIT-SIPI-SIPI Sequence and Choice of Timeouts 10-24
Table 10-2. Initial APIC IDs for the Logical Processors in a System that has Four Intel Xeon MP Processors Supporting Intel Hyper-Threading Technology[1] 10-40
Table 10-3. Initial APIC IDs for the Logical Processors in a System that has Two Physical Processors Supporting Dual-Core and Intel Hyper-Threading Technology 10-41
Table 10-4. Example of Possible x2APIC ID Assignment in a System that has Two Physical Processors Supporting x2APIC and Intel Hyper-Threading Technology 10-41
Table 10-5. Boot Phase IPI Message Format 10-56
Table 11-1. IA-32 and Intel® 64 Processor States Following Power-up, Reset, or INIT 11-2
Table 11-2. Variance of RESET Values in Selected Intel Architecture Processors 11-4
Table 11-3. Recommended Settings of EM and MP Flags on IA-32 Processors 11-6
Table 11-4. Software Emulation Settings of EM, MP, and NE Flags 11-7
Table 11-5. Main Initialization Steps in STARTUP.ASM Source Listing 11-16
Table 11-6. Relationship Between BLD Item and ASM Source File 11-27
Table 11-7. Microcode Update Field Definitions 11-28
Table 11-8. Microcode Update Format 11-30
Table 11-9. Extended Processor Signature Table Header Structure 11-31

PAGE

Table 11-10. Processor Signature Structure 11-31
Table 11-11. Processor Flags....11-33
Table 11-12. Microcode Update Signature 11-37
Table 11-13. Microcode Update Functions 11-42
Table 11-14. Parameters for the Presence Test....11-42
Table 11-15. Parameters for the Write Update Data Function....11-43
Table 11-16. Parameters for the Control Update Sub-function 11-47
Table 11-17. Mnemonic Values....11-47
Table 11-18. Parameters for the Read Microcode Update Data Function....11-47
Table 11-19. Return Code Definitions....11-49
Table 12-1. Local APIC Register Address Map 12-6
Table 12-2. Local APIC Timer Modes....12-17
Table 12-3. Valid Combinations for Pentium 4 and Intel Xeon Processors Local xAPIC Interrupt Command Register....12-21
Table 12-4. Valid Combinations for the P6 Family Processor Local APIC Interrupt Command Register 12-22
Table 12-5. x2APIC Operating Mode Configurations 12-37
Table 12-6. Local APIC Register Address Map Supported by x2APIC....12-38
Table 12-7. MSR/MMIO Interface of a Local x2APIC in Different Modes of Operation 12-40
Table 12-8. EOI Message (14 Cycles) 12-47
Table 12-9. Short Message (21 Cycles) 12-48
Table 12-10. Non-Focused Lowest Priority Message (34 Cycles) 12-49
Table 12-11. APIC Bus Status Cycles Interpretation 12-51
Table 13-1. Characteristics of the Caches, TLBs, Store Buffer, and Write Combining Buffer in Intel 64 and IA-32 Processors . 13-2
Table 13-2. Memory Types and Their Properties 13-6
Table 13-3. Methods of Caching Available in Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, P6 Family, and Pentium Processors 13-7
Table 13-4. MESI Cache Line States 13-9
Table 13-5. Cache Operating Modes 13-12
Table 13-6. Effective Page-Level Memory Type for Pentium Pro and Pentium II Processors 13-14
Table 13-7. Effective Page-Level Memory Types for Pentium III and More Recent Processor Families....13-15
Table 13-8. Memory Types That Can Be Encoded in MTRRs....13-21
Table 13-9. Address Mapping for Fixed-Range MTRRs....13-24
Table 13-10. Memory Types That Can Be Encoded With PAT....13-34
Table 13-11. Selection of PAT Entries with PAT, PCD, and PWT Flags....13-35
Table 13-12. Memory Type Setting of PAT Entries Following a Power-up or Reset....13-35
Table 14-1. Action Taken By MMX Instructions for Different Combinations of EM, MP, and TS 14-1
Table 14-3. Effect of the MMX, x87 FPU, and FXSAVE/FXRSTOR Instructions on the x87 FPU Tag Word....14-3
Table 14-2. Effects of MMX Instructions on x87 FPU State 14-3
Table 15-1. Action Taken for Combinations of OSFXSR, OSXMMEXCPT, SSE, SSE2, SSE3, EM, MP, and TS....15-3
Table 15-2. Action Taken for Combinations of OSFXSR, SSSE3, SSE4, EM, and TS 15-4
Table 15-3. CPUID.(EAX=0DH, ECX=1) EAX Bit Assignment....15-8
Table 16-1. Architectural and Non-Architectural MSRs Related to HWP....16-6
Table 16-2. IA32_HWP_CTL MSR Bit 0 Behavior 16-13
Table 16-3. Architectural and non-Architecture MSRs Related to HDC....16-21
Table 16-4. Hardware Feedback Interface Structure 16-25
Table 16-5. Hardware Feedback Interface Global Header Structure 16-26
Table 16-6. Hardware Feedback Interface Logical Processor Entry Structure 16-26
Table 16-7. Intel® Thread Director Table Structure 16-27
Table 16-8. Intel® Thread Director Global Header Structure 16-28
Table 16-9. Intel® Thread Director Logical Processor Entry Structure....16-29
Table 16-10. IA32_HW_FEEDBACK_CONFIG Control Options....16-32
Table 16-11. On-Demand Clock Modulation Duty Cycle Field Encoding 16-40
Table 16-12. RAPL MSR Interfaces and RAPL Domains....16-50
Table 17-1. Bits 54:53 in IA32_MCi_STATUS MSRs when IA32_MCG_CAP[11] = 1 and UC = 0 17-7
Table 17-2. Overwrite Rules for Enabled Errors 17-8
Table 17-3. Address Mode in IA32_MCi_MISC[8:6]....17-10
Table 17-4. Address Mode in IA32_MCi_MISC[8:6]....17-11
Table 17-5. Extended Machine Check State MSRs in Processors Without Support for Intel® 64 Architecture 17-12
Table 17-6. Extended Machine Check State MSRs In Processors With Support for Intel® 64 Architecture 17-12
Table 17-7. MC Error Classifications 17-18
Table 17-8. Overwrite Rules for UC, CE, and UCR Errors 17-18
Table 17-9. IA32_MCi_Status [15:0] Simple Error Code Encoding....17-21
Table 17-10. IA32_MCi_Status [15:0] Compound Error Code Encoding....17-21
Table 17-11. Encoding for TT (Transaction Type) Sub-Field 17-22
Table 17-12. Level Encoding for LL (Memory Hierarchy Level) Sub-Field 17-22
Table 17-13. Encoding of Request (RRRR) Sub-Field....17-23

PAGE

Table 17-14. Encodings of PP, T, and II Sub-Fields 17-23
Table 17-16. MCA Compound Error Code Encoding for SRAO Errors 17-24
Table 17-17. IA32_MCi_STATUS Values for SRAO Errors 17-24
Table 17-15. Encodings of MMM and CCCC Sub-Fields 17-24
Table 17-18. IA32_MCG_STATUS Flag Indication for SRAO Errors 17-25
Table 17-20. IA32_MCi_STATUS Values for All Defined SRAR Errors 17-26
Table 17-19. MCA Compound Error Code Encoding for SRAR Errors 17-26
Table 17-21. IA32_MCG_STATUS Flag Indication for SRAR Errors 17-27
Table 18-1. CPUID DisplayFamily_DisplayModel Signatures for Processor Family 06H 18-1
Table 18-2. Incremental Decoding Information: Processor Family 06H Machine Error Codes for Machine Check 18-1
Table 18-3. CPUID DisplayFamily_DisplayModel Signatures for Processors Based on Intel® Core™ Microarchitecture 18-3
Table 18-4. Incremental Bus Error Codes of Machine Check for Processors Based on Intel® Core™ Microarchitecture 18-4
Table 18-5. Incremental MCA Error Code Types for Intel® Xeon® Processor 7400 18-6
Table 18-6. Type B: Bus and Interconnect Error Codes 18-6
Table 18-7. Type C: Cache Bus Controller Error Codes 18-7
Table 18-8. Intel® QPI Machine Check Error Codes for IA32_MC0_STATUS and IA32_MC1_STATUS 18-8
Table 18-9. Intel® QPI Machine Check Error Codes for IA32_MC0_MISC and IA32_MC1_MISC 18-8
Table 18-10. Machine Check Error Codes for IA32_MC7_STATUS 18-9
Table 18-11. Incremental Memory Controller Error Codes of Machine Check for IA32_MC8_STATUS 18-9
Table 18-12. Incremental Memory Controller Error Codes of Machine Check for IA32_MC8_MISC 18-10
Table 18-13. Machine Check Error Codes for IA32_MC4_STATUS 18-10
Table 18-14. Intel® QPI MC Error Codes for IA32_MC6_STATUS and IA32_MC7_STATUS 18-11
Table 18-15. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 8, 11) 18-12
Table 18-16. Intel IMC MC Error Codes for IA32_MCi_MISC (i= 8, 11) 18-12
Table 18-17. Machine Check Error Codes for IA32_MC4_STATUS 18-13
Table 18-18. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9—16) 18-14
Table 18-19. Intel IMC MC Error Codes for IA32_MCi_MISC (i= 9—16) 18-15
Table 18-20. Machine Check Error Codes for IA32_MC4_STATUS 18-16
Table 18-21. Intel® QPI MC Error Codes for IA32_MCi_STATUS (i = 5, 20, 21) 18-17
Table 18-22. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9—16) 18-18
Table 18-23. Intel IMC MC Error Codes for IA32_MCi_MISC (i= 9—16) 18-18
Table 18-24. Machine Check Error Codes for IA32_MC4_STATUS 18-19
Table 18-25. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9—10) 18-20
Table 18-26. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 9—16) 18-21
Table 18-27. Intel HA MC Error Codes for IA32_MCi_MISC (i= 7, 8) 18-22
Table 18-28. Machine Check Error Codes for IA32_MC4_STATUS 18-22
Table 18-29. Interconnect MC Error Codes for IA32_MCi_STATUS (i = 5, 12, 19) 18-24
Table 18-30. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13—18) 18-26
Table 18-31. M2M MC Error Codes for IA32_MCi_STATUS (i= 7, 8) 18-27
Table 18-32. Intel HA MC Error Codes for IA32_MCi_MISC (i= 7, 8) 18-27
Table 18-33. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 7, 8) 18-28
Table 18-34. Machine Check Error Codes for IA32_MC4_STATUS 18-29
Table 18-35. Interconnect MC Error Codes for IA32_MCi_STATUS (i = 5, 7, 8) 18-31
Table 18-36. MSRs Reporting MC Error Codes by CPUID DisplayFamily_DisplaySignature 18-32
Table 18-37. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13—14, 17—18, 21—22, 25—26) 18-33
Table 18-38. Additional Information Reported in IA32_MCi_MISC (i= 13—14, 17—18, 21—22, 25—26) 18-35
Table 18-39. M2M MC Error Codes for IA32_MCi_STATUS (i= 12, 16, 20, 24) 18-36
Table 18-40. Machine Check Error Codes for IA32_MC4_STATUS 18-37
Table 18-41. Interconnect MC Error Codes for IA32_MC5_STATUS 18-40
Table 18-42. Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13—20) 18-41
Table 18-43. Additional Information Reported in IA32_MCi_MISC (i= 13—20) 18-43
Table 18-44. M2M MC Error Codes for IA32_MC12_STATUS 18-44
Table 18-45. Incremental Decoding Information: Processor Family 0FH, Machine Error Codes for Machine Check 18-45
Table 18-46. MCi_STATUS Register Bit Definition 18-46
Table 18-47. Incremental MCA Error Code for Intel® Xeon® Processor MP 7100 18-47
Table 18-48. Other Information Field Bit Definition 18-47
Table 18-49. Type A: L3 Error Codes 18-48
Table 18-50. Type B: Bus and Interconnect Error Codes 18-48
Table 18-51. Type C: Cache Bus Controller Error Codes 18-49
Table 18-52. Decoding Family 0FH Machine Check Codes for Cache Hierarchy Errors 18-50
Table 19-1. Breakpoint Examples 19-6
Table 19-2. Debug Exception Conditions 19-9
Table 19-4. LBR Stack Size and TOS Pointer Range 19-17
Table 19-3. Legacy and Streamlined Operation with Freeze_PerfMon_On_PMI = 1, Counter Overflowed 19-17
Table 19-5. IA32_DEBUGCTL Flag Encodings 19-25

PAGE

Table 19-6. CPL-Qualified Branch Trace Store Encodings 19-26
Table 19-7. MSR_LASTBRANCH_x_TO_IP for the Goldmont Microarchitecture 19-28
Table 19-8. MSR_LASTBRANCH_x_FROM_IP 19-30
Table 19-9. MSR_LASTBRANCH_x_TO_IP 19-31
Table 19-10. LBR Stack Size and TOS Pointer Range 19-31
Table 19-11. MSR_LBR_SELECT for Nehalem Microarchitecture 19-31
Table 19-12. MSR_LBR_SELECT for Sandy Bridge Microarchitecture 19-32
Table 19-13. MSR_LBR_SELECT for Haswell Microarchitecture 19-32
Table 19-14. MSR_LASTBRANCH_x_FROM_IP with TSX Information 19-33
Table 19-15. LBR Stack Size and TOS Pointer Range 19-34
Table 19-16. MSR_LBR_INFO_x 19-34
Table 19-17. LBR MSR Stack Size and TOS Pointer Range for the Pentium® 4 and the Intel® Xeon® Processor Family 19-37
Table 19-18. Monitoring Supported Event IDs 19-49
Table 19-19. Re-indexing of CLOS Numbers and Mapping to CAT/CDP Mask MSRs 19-63
Table 19-20. MBA Delay Value MSRs 19-68
Table 20-1. LBR IP Values for Various Operations 20-2
Table 20-2. Branch Type Filtering Details 20-3
Table 20-3. IA32_LBR_x_INFO and IA32_LER_INFO Branch Type Encodings 20-4
Table 20-4. LBR VMCS Fields 20-6
Table 21-1. Association of Fixed-Function Performance Counters with Architectural Performance Events 21-7
Table 21-2. New Performance Monitoring MSR Naming Details 21-16
Table 21-3. UMask and Event Select Encodings for Pre-Defined Architectural Performance Events 21-18
Table 21-4. PEBS Record Format for Intel Core i7 Processor Family 21-25
Table 21-5. Data Source Encoding for Load Latency Record 21-29
Table 21-6. Off-Core Response Event Encoding 21-30
Table 21-7. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition 21-30
Table 21-8. Opcode Field Encoding for MSR_UNCORE_ADDR_OPCODE_MATCH 21-35
Table 21-9. Uncore PMU MSR Summary 21-37
Table 21-10. Uncore PMU MSR Summary for Intel® Xeon® Processor E7 Family 21-38
Table 21-11. Core PMU Comparison 21-39
Table 21-12. PEBS Facility Comparison 21-42
Table 21-13. PEBS Performance Events for Sandy Bridge Microarchitecture 21-43
Table 21-14. Layout of Data Source Field of Load Latency Record 21-45
Table 21-16. Off-Core Response Event Encoding 21-46
Table 21-15. Layout of Precise Store Information In PEBS Record 21-46
Table 21-17. MSR_OFFCORE_RSP_x Request_Type Field Definition 21-47
Table 21-18. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition 21-48
Table 21-19. MSR_OFFCORE_RSP_x Snoop Info Field Definition 21-49
Table 21-20. Uncore PMU MSR Summary 21-50
Table 21-21. MSR_OFFCORE_RSP_x Supplier Info Field Definitions 21-51
Table 21-22. Uncore PMU MSR Summary for Intel® Xeon® Processor E5 Family 21-52
Table 21-23. Core PMU Comparison 21-53
Table 21-24. PEBS Facility Comparison 21-53
Table 21-25. PEBS Record Format for 4th Generation Intel Core Processor Family 21-54
Table 21-26. Precise Events That Supports Data Linear Address Profiling 21-55
Table 21-27. Layout of Data Linear Address Information In PEBS Record 21-56
Table 21-28. MSR_OFFCORE_RSP_x Request_Type Definition (Haswell Microarchitecture) 21-56
Table 21-29. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signatures: 06_3CH, 06_46H) 21-57
Table 21-30. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature: 06_45H) 21-57
Table 21-31. MSR_OFFCORE_RSP_x Supplier Info Field Definition 21-58
Table 21-32. TX Abort Information Field Definition 21-60
Table 21-33. Uncore PMU MSR Summary 21-60
Table 21-34. Core PMU Comparison 21-62
Table 21-35. PEBS Facility Comparison 21-63
Table 21-36. PEBS Record Format for the 6th Generation, 7th Generation, and 8th Generation Intel Core Processor Families 21-64
Table 21-37. Precise Events for the Skylake, Kaby Lake, and Coffee Lake Microarchitectures 21-65
Table 21-38. Layout of Data Linear Address Information In PEBS Record 21-66
Table 21-39. FrontEnd_Retired Sub-Event Encodings Supported by MSR_PEBS_FRONTEND.EVTSEL 21-67
Table 21-40. MSR_PEBS_FRONTEND Layout 21-67
Table 21-41. MSR_OFFCORE_RSP_x Request_Type Definition (Skylake, Kaby Lake, and Coffee Lake Microarchitectures) 21-68
Table 21-42. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signatures: 06_4EH, 06_5EH, 06_8EH, 06_9EH) 21-68
Table 21-43. MSR_OFFCORE_RSP_x Snoop Info Field Definition (CPUID Signatures: 06_4EH, 06_5EH, 06_8EH, 06_9E, 06_55H) 21-69
Table 21-45. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature: 06_55H) 21-70
Table 21-44. MSR_OFFCORE_RSP_x Request_Type Definition (Intel® Xeon® Scalable Processor Family) 21-70

PAGE

Table 21-46. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature: 06_55H, Steppings 0x5H - 0xFH) 21-71
Table 21-48. MSR_OFFCORE_RSP_x Request_Type Definition (Processors Based on Ice Lake Microarchitecture) 21-72
Table 21-47. Core PMU Summary of the Ice Lake Microarchitecture 21-72
Table 21-49. MSR_OFFCORE_RSP_x Supplier Info Field Definition (Processors Based on Ice Lake Microarchitecture) 21-73
Table 21-50. MSR_OFFCORE_RSP_x Snoop Info Field Definition (Processors Based on Ice Lake Microarchitecture) 21-74
Table 21-51. Core PMU Summary of the Golden Cove Microarchitecture 21-76
Table 21-52. Special Performance Monitoring Events with Counter Restrictions 21-77
Table 21-53. Core PMU Summary of the Gracemont Microarchitecture 21-78
Table 21-54. E-core PEBS Memory Access Info Encoding 21-78
Table 21-55. E-core PEBS Data Source Encodings 21-79
Table 21-56. MSR_OFFCORE_RSPx Request Type Definition 21-80
Table 21-57. Core PMU Summary of the Lion Cove Microarchitecture 21-81
Table 21-58. Performance Monitoring Events with Counter Restrictions in Lion Cove PMU 21-82
Table 21-59. Core PMU Summary of the Skymont Microarchitecture 21-82
Table 21-60. PEBS Performance Events for Knights Landing Microarchitecture 21-84
Table 21-61. PEBS Record Format for Knights Landing Microarchitecture 21-84
Table 21-62. OffCore Response Event Encoding 21-85
Table 21-63. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers 21-85
Table 21-64. PEBS Performance Events for the Silvermont Microarchitecture 21-88
Table 21-66. OffCore Response Event Encoding 21-89
Table 21-65. PEBS Record Format for the Silvermont Microarchitecture 21-89
Table 21-67. MSR_OFFCORE_RSPx Request_Type Field Definition 21-90
Table 21-68. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition 21-91
Table 21-69. MSR_OFFCORE_RSPx Snoop Info Field Definition 21-91
Table 21-70. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures 21-92
Table 21-71. Precise Events Supported by the Goldmont Microarchitecture 21-94
Table 21-72. PEBS Record Format for the Goldmont Microarchitecture 21-95
Table 21-73. MSR_OFFCORE_RSPx Request_Type Field Definition 21-96
Table 21-74. MSR_OFFCORE_RSPx For L2 Miss and Outstanding Requests 21-97
Table 21-75. Core PMU Comparison Between the Goldmont Plus and Goldmont Microarchitectures 21-98
Table 21-76. Core PMU Comparison Between the Tremont and Goldmont Plus Microarchitectures 21-99
Table 21-77. New Fields in IA32_PEBS_ENABLE 21-100
Table 21-78. MSR_OFFCORE_RSPx Request Type Definition 21-101
Table 21-79. MSR_OFFCORE_RSPx Response Type Definition 21-102
Table 21-80. MSR_OFFCORE_RSPx Snoop Info Definition 21-102
Table 21-81. Core Specificity Encoding within a Non-Architectural Umask 21-103
Table 21-82. Agent Specificity Encoding within a Non-Architectural Umask 21-104
Table 21-83. HW Prefetch Qualification Encoding within a Non-Architectural Umask 21-104
Table 21-84. MESI Qualification Definitions within a Non-Architectural Umask 21-104
Table 21-85. Bus Snoop Qualification Definitions within a Non-Architectural Umask 21-105
Table 21-86. Snoop Type Qualification Definitions within a Non-Architectural Umask 21-105
Table 21-87. At-Retirement Performance Events for Intel Core Microarchitecture 21-108
Table 21-88. PEBS Performance Events for Intel Core Microarchitecture 21-109
Table 21-89. Requirements to Program PEBS 21-110
Table 21-90. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) 21-111
Table 21-91. Event Example 21-118
Table 21-92. CCR Names and Bit Positions 21-122
Table 21-93. Effect of Logical Processor and CPL Qualification for Logical-Processor-Specific (TS) Events 21-130
Table 21-94. Effect of Logical Processor and CPL Qualification for Non-logical-Processor-specific (TI) Events 21-131
Table 21-95. Nominal Core Crystal Clock Frequency 21-147
Table 21-96. Basic Info Group 21-153
Table 21-97. Memory Access Info Group 21-154
Table 21-98. Updated Memory Access Info Group 21-155
Table 21-99. GPRs in Ice Lake Microarchitecture 21-156
Table 21-100. XMMs 21-156
Table 21-101. LBRs 21-157
Table 21-102. MSR_PEBS_CFG Programming 21-158
Table 21-103. Counters Group 21-159
Table 21-104. PEBS Record Example 1 21-160
Table 21-105. PEBS Record Example 2 21-162
Table 21-106. Data Source Encoding for Memory Accesses (Ice Lake and Later Microarchitectures) 21-164
Table 21-107. Data Source Encoding for Memory Accesses (Lion Cove and Next Generation Microarchitectures) 21-164
Table 21-108. PEBS Basic Info Group 21-166
Table 22-1. Real-Address Mode Exceptions and Interrupts 22-6

PAGE

Table 22-2. Software Interrupt Handling Methods While in Virtual-8086 Mode 22-17
Table 23-1. Characteristics of 16-Bit and 32-Bit Program Modules 23-1
Table 24-1. New Instruction in the Pentium Processor and Later IA-32 Processors 24-4
Table 24-3. EM and MP Flag Interpretation 24-17
Table 24-2. Recommended Values of the EM, MP, and NE Flags for Intel486 SX Microprocessor/Intel 487 SX Math Coprocessor System. 24-17
Table 24-4. Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment 24-22
Table 24-5. Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception 24-23
Table 24-6. Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception 24-24
Table 24-7. Exception Conditions for SIMD/MMX Instructions with Memory Reference 24-25
Table 24-8. Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception 24-26
Table 24-9. Exception Conditions for Legacy SIMD/MMX Instructions without Memory Reference 24-27
Table 24-10. Processor State Following Power-up/Reset/INIT for Pentium, Pentium Pro and Pentium 4 Processors. 24-37
Table 26-1. Format of the VMCS Region 26-2
Table 26-2. Format of Access Rights 26-4
Table 26-3. Format of Interruptibility State 26-6
Table 26-4. Format of Pending-Debug-Exceptions. 26-7
Table 26-5. Definitions of Pin-Based VM-Execution Controls 26-10
Table 26-6. Definitions of Primary Processor-Based VM-Execution Controls 26-10
Table 26-7. Definitions of Secondary Processor-Based VM-Execution Controls. 26-12
Table 26-8. Definitions of Tertiary Processor-Based VM-Execution Controls 26-13
Table 26-9. Format of Extended-Page-Table Pointer 26-17
Table 26-10. Definitions of VM-Function Controls 26-18
Table 26-11. Format of Sub-Page-Permission-Table Pointer 26-19
Table 26-12. Format of Hypervisor-Managed Linear-Address Translation Pointer 26-20
Table 26-13. Definitions of Primary VM-Exit Controls 26-21
Table 26-14. Definitions of Secondary VM-Exit Controls 26-22
Table 26-15. Format of an MSR Entry. 26-23
Table 26-16. Definitions of VM-Entry Controls 26-23
Table 26-17. Format of the VM-Entry Interruption-Information Field 26-25
Table 26-18. Format of Exit Reason 26-26
Table 26-19. Format of the VM-Exit Interruption-Information Field 26-27
Table 26-20. Format of the IDT-Vectoring Information Field 26-28
Table 26-21. Structure of VMCS Component Encoding 26-30
Table 27-1. Format of the Virtualization-Exception Information Area 27-22
Table 29-1. Exit Qualification for Debug Exceptions 29-4
Table 29-2. Exit Qualification for Task Switches 29-5
Table 29-3. Exit Qualification for Control-Register Accesses 29-7
Table 29-4. Exit Qualification for MOV DR 29-8
Table 29-5. Exit Qualification for I/O Instructions. 29-9
Table 29-6. Exit Qualification for APIC-Access VM Exits from Linear Accesses and Guest-Physical Accesses 29-9
Table 29-7. Exit Qualification for EPT Violations 29-11
Table 29-8. Format of the VM-Exit Instruction-Information Field as Used for INS and OUTS 29-16
Table 29-9. Format of the VM-Exit Instruction-Information Field as Used for INVEPT, INVPCID, and INVVPID 29-17
Table 29-10. Format of the VM-Exit Instruction-Information Field as Used for LIDT, LGDT, SIDT, or SGDT 29-18
Table 29-11. Format of the VM-Exit Instruction-Information Field as Used for LLDT, LTR, SLDT, and STR 29-19
Table 29-12. Format of the VM-Exit Instruction-Information Field as Used for RDRAND and RDSEED 29-20
Table 29-13. Format of the VM-Exit Instruction-Information Field as Used for TPAUSE and UMWAIT 29-20
Table 29-14. Format of the VM-Exit Instruction-Information Field as Used for VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, and XSAVES 29-21
Table 29-15. Format of the VM-Exit Instruction-Information Field as Used for VMREAD and VMWRITE 29-22
Table 29-16. Format of the VM-Exit Instruction-Information Field as Used for LOADIWKEY 29-23
Table 30-1. Format of an EPT PML5 Entry (PML5E) that References an EPT PML4 Table 30-4
Table 30-2. Format of an EPT PML4 Entry (PML4E) that References an EPT Page-Directory-Pointer Table 30-5
Table 30-3. Format of an EPT Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page 30-6
Table 30-4. Format of an EPT Page-Directory-Pointer-Table Entry (PDPTE) that References an EPT Page Directory 30-7
Table 30-5. Format of an EPT Page-Directory Entry (PDE) that Maps a 2-MByte Page 30-8
Table 30-6. Format of an EPT Page-Directory Entry (PDE) that References an EPT Page Table 30-9
Table 30-7. Format of an EPT Page-Table Entry that Maps a 4-KByte Page 30-11
Table 31-1. Format of Posted-Interrupt Descriptor (PID). 31-4
Table 31-2. Format of Posted-Interrupt Descriptor (PID). 31-16
Table 32-1. VM-Instruction Error Numbers 32-31
Table 33-1. SMRAM State Save Map. 33-5
Table 33-2. Processor Signatures and 64-bit SMRAM State Save Map Format 33-6
Table 33-3. SMRAM State Save Map for Intel 64 Architecture 33-7

PAGE

Table 33-4. Processor Register Initialization in SMM. 33-9
Table 33-5. I/O Instruction Information in the SMM State Save Map 33-12
Table 33-6. I/O Instruction Type Encodings 33-12
Table 33-7. Auto HALT Restart Flag Values 33-14
Table 33-8. I/O Instruction Restart Field Values. 33-16
Table 33-9. Exit Qualification for SMIs That Arrive Immediately After the Retirement of an I/O Instruction. 33-21
Table 33-10. Format of MSEG Header 33-25
Table 34-1. COFI Type for Branch Instructions 34-3
Table 34-2. IP Filtering Packet Example 34-7
Table 34-3. ToPA Table Entry Fields 34-12
Table 34-4. Algorithm to Manage Intel PT ToPA PMI and XSAVES/XRSTORS. 34-15
Table 34-5. Behavior on Restricted Memory Access. 34-16
Table 34-6. IA32_RTIT_CTL MSR. 34-17
Table 34-7. IA32_RTIT_STATUS MSR 34-21
Table 34-8. IA32_RTIT_OUTPUT_BASE MSR. 34-23
Table 34-10. TSX Packet Scenarios with BranchEn=1 34-24
Table 34-9. IA32_RTIT_OUTPUT_MASK_PTRS MSR. 34-24
Table 34-11. CPUID Leaf 14H Enumeration of Intel Processor Trace Capabilities 34-27
Table 34-12. CPUID Leaf 14H, sub-leaf 1H Enumeration of Intel Processor Trace Capabilities. 34-29
Table 34-13. An Illustrative CYC Packet Example. 34-34
Table 34-14. Compound Packet Event Summary 34-37
Table 34-15. Packets Forbidden Between BBP and BEP 34-37
Table 34-16. TNT Packet Definition 34-39
Table 34-18. FUP/TIP IP Reconstruction 34-41
Table 34-19. TNT Examples with Deferred TIPs. 34-42
Table 34-20. TIP.PGE Packet Definition 34-43
Table 34-21. TIP.PGD Packet Definition. 34-44
Table 34-22. FUP Packet Definition 34-45
Table 34-23. FUP Cases and IP Payload. 34-46
Table 34-24. PIP Packet Definition 34-47
Table 34-25. General Form of MODE Packets 34-48
Table 34-26. MODE.Exec Packet Definition. 34-48
Table 34-27. MODE.TSX Packet Definition 34-49
Table 34-28. TraceStop Packet Definition 34-50
Table 34-29. CBR Packet Definition 34-50
Table 34-30. TSC Packet Definition. 34-51
Table 34-31. MTC Packet Definition 34-52
Table 34-32. TMA Packet Definition 34-53
Table 34-33. Cycle Count Packet Definition 34-54
Table 34-34. VMCS Packet Definition. 34-55
Table 34-35. OVF Packet Definition 34-56
Table 34-36. PSB Packet Definition 34-56
Table 34-37. PSBEND Packet Definition 34-57
Table 34-38. MNT Packet Definition. 34-58
Table 34-39. PAD Packet Definition 34-58
Table 34-40. PTW Packet Definition. 34-59
Table 34-41. EXSTOP Packet Definition 34-60
Table 34-42. MWAIT Packet Definition. 34-61
Table 34-43. PWRE Packet Definition. 34-62
Table 34-44. PWRX Packet Definition 34-63
Table 34-45. Block Begin Packet Definition 34-64
Table 34-46. Block Item Packet Definition 34-65
Table 34-47. BIP Encodings 34-65
Table 34-48. Block End Packet Definition 34-70
Table 34-49. Control Flow Event Packet Definition. 34-71
Table 34-50. CFE Packet Type and Vector Fields Details. 34-71
Table 34-51. Event Data Packet Definition. 34-73
Table 34-52. VMX Controls For Intel Processor Trace 34-74
Table 34-53. Packets on VMX Transitions (System-Wide Tracing) 34-75
Table 34-54. Packets on a Failed VM Entry 34-76
Table 34-55. Packet Generation under Different Example Operations 34-77
Table 34-56. Packet Generation with Operations That Alter the Value of PacketEn 34-78
Table 34-57. Examples of PTWRITE when TriggerEn && PTWEn is True 34-79
Table 34-58. Examples of Power Event Trace when TriggerEn && PwrEvtEn is True. 34-79
Table 34-59. Event Trace Examples when TriggerEn && ContextEn && EventEn is True 34-80

PAGE

Table 35-1. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1 35-3
Table 35-2. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX2 35-4
Table 35-3. VMX Operation and Supervisor Mode Enclave Instruction Leaf Functions in Long-Form of OVERSUB 35-4
Table 35-4. Intel® SGX Opt-in and Enabling Behavior 35-5
Table 35-5. CPUID Leaf 12H, Sub-Leaf 0 Enumeration of Intel® SGX Capabilities 35-5
Table 35-6. CPUID Leaf 12H, Sub-Leaf 1 Enumeration of Intel® SGX Capabilities 35-6
Table 35-7. CPUID Leaf 12H, Sub-Leaf Index 2 or Higher Enumeration of Intel® SGX Resources 35-6
Table 36-1. List of Implicit and Explicit Memory Access by Intel® SGX Enclave Instructions 36-3
Table 36-2. Layout of SGX Enclave Control Structure (SECS) 36-5
Table 36-3. Layout of ATTRIBUTES Structure 36-6
Table 36-4. Bit Vector Layout of MISCSELECT Field of Extended Information 36-6
Table 36-5. Bit Vector Layout of CET_ATTRIBUTES Field of Extended Information 36-7
Table 36-6. Layout of Thread Control Structure (TCS) 36-7
Table 36-7. Layout of TCS.FLAGS Field 36-8
Table 36-8. Top-to-Bottom Layout of an SSA Frame 36-9
Table 36-9. Layout of GPRSGX Portion of the State Save Area 36-9
Table 36-10. Layout of EXITINFO Field 36-10
Table 36-11. Exception Vectors 36-10
Table 36-12. Layout of MISC region of the State Save Area 36-11
Table 36-13. Layout of EXINFO Structure 36-11
Table 36-14. Layout of CET State Save Area Frame 36-12
Table 36-15. Layout of PAGEINFO Data Structure 36-12
Table 36-16. Layout of SECINFO Data Structure 36-12
Table 36-17. Layout of SECINFO.FLAGS Field 36-13
Table 36-18. Supported PAGE_TYPE 36-13
Table 36-19. Layout of PCMD Data Structure 36-14
Table 36-20. Layout of Enclave Signature Structure (SIGSTRUCT) 36-14
Table 36-22. Layout of REPORT 36-16
Table 36-21. Layout of EINIT Token (EINITTOKEN) 36-16
Table 36-23. Layout of TARGETINFO Data Structure 36-17
Table 36-24. Layout of KEYREQUEST Data Structure 36-18
Table 36-25. Supported KEYName Values 36-18
Table 36-26. Layout of KEYPOLICY Field 36-18
Table 36-27. Layout of Version Array Data Structure 36-19
Table 36-28. Content of an Enclave Page Cache Map Entry 36-19
Table 36-29. Layout of RDINFO Structure 36-20
Table 36-30. Layout of RDINFO STATUS Structure 36-20
Table 36-31. Layout of RDINFO FLAGS Structure 36-20
Table 37-1. Illegal Instructions Inside an Enclave 37-15
Table 38-1. GPR, x87 Synthetic States on Asynchronous Enclave Exit 38-3
Table 39-1. Register Usage of Privileged Enclave Instruction Leaf Functions 39-1
Table 39-2. Register Usage of Unprivileged Enclave Instruction Leaf Functions 39-2
Table 39-3. Register Usage of Virtualization Operation Enclave Instruction Leaf Functions 39-2
Table 39-4. Error or Information Codes for Intel® SGX Instructions 39-2
Table 39-5. List of Internal CREG 39-3
Table 39-6. Base Concurrency Restrictions 39-5
Table 39-7. Additional Concurrency Restrictions 39-7
Table 39-8. Base Concurrency Restrictions of EADD 39-17
Table 39-9. Additional Concurrency Restrictions of EADD 39-18
Table 39-10. Base Concurrency Restrictions of EAUG 39-22
Table 39-11. Additional Concurrency Restrictions of EAUG 39-23
Table 39-12. EBLOCK Return Value in RAX 39-27
Table 39-13. Base Concurrency Restrictions of EBLOCK 39-27
Table 39-14. Additional Concurrency Restrictions of EBLOCK 39-28
Table 39-15. Base Concurrency Restrictions of ECREATE 39-30
Table 39-16. Additional Concurrency Restrictions of ECREATE 39-31
Table 39-17. EDBGRD Return Value in RAX 39-36
Table 39-18. Base Concurrency Restrictions of EDBGRD 39-37
Table 39-19. Additional Concurrency Restrictions of EDBGRD 39-37
Table 39-20. EDBGWR Return Value in RAX 39-40
Table 39-21. Base Concurrency Restrictions of EDBGWR 39-41
Table 39-22. Additional Concurrency Restrictions of EDBGWR 39-41
Table 39-23. Base Concurrency Restrictions of EEXTEND 39-44
Table 39-24. Additional Concurrency Restrictions of EEXTEND 39-45
Table 39-25. EINIT Return Value in RAX 39-48

PAGE

Table 39-26. Base Concurrency Restrictions of EINIT 39-49
Table 39-27. Additional Concurrency Restrictions of ENIT 39-49
Table 39-28. ELDB/ELDU/ELDBC/ELBUC Return Value in RAX 39-55
Table 39-29. Base Concurrency Restrictions of ELDB/ELDU/ELDBC/ELBUC 39-56
Table 39-30. Additional Concurrency Restrictions of ELDB/ELDU/ELDBC/ELBUC 39-56
Table 39-31. EMODPR Return Value in RAX 39-61
Table 39-32. Base Concurrency Restrictions of EMODPR 39-61
Table 39-33. Additional Concurrency Restrictions of EMODPR 39-62
Table 39-34. EMODT Return Value in RAX 39-64
Table 39-35. Base Concurrency Restrictions of EMODT 39-64
Table 39-36. Additional Concurrency Restrictions of EMODT 39-65
Table 39-37. Base Concurrency Restrictions of EPA 39-67
Table 39-38. Additional Concurrency Restrictions of EPA 39-67
Table 39-39. ERDINFO Return Value in RAX 39-69
Table 39-40. Base Concurrency Restrictions of ERDINFO 39-70
Table 39-41. Additional Concurrency Restrictions of ERDINFO 39-70
Table 39-42. EREMOVE Return Value in RAX 39-73
Table 39-43. Base Concurrency Restrictions of EREMOVE 39-74
Table 39-44. Additional Concurrency Restrictions of EREMOVE 39-74
Table 39-45. ETRACK Return Value in RAX 39-77
Table 39-46. Base Concurrency Restrictions of ETRACK 39-77
Table 39-47. Additional Concurrency Restrictions of ETRACK 39-77
Table 39-48. ETRACKC Return Value in RAX 39-80
Table 39-49. Base Concurrency Restrictions of ETRACKC 39-80
Table 39-50. Additional Concurrency Restrictions of ETRACKC 39-81
Table 39-51. EUPDATESVN Return Value in RAX 39-84
Table 39-52. Base Concurrency Restrictions of EUPDATESVN 39-84
Table 39-53. Additional Concurrency Restrictions of EUPDATESVN 39-84
Table 39-54. EWB Return Value in RAX 39-86
Table 39-55. Base Concurrency Restrictions of EWB 39-86
Table 39-56. Additional Concurrency Restrictions of EWB 39-86
Table 39-57. EACCEPT Return Value in RAX 39-92
Table 39-58. Base Concurrency Restrictions of EACCEPT 39-93
Table 39-59. Additional Concurrency Restrictions of EACCEPT 39-93
Table 39-60. EACCEPTCOPY Return Value in RAX 39-97
Table 39-61. Base Concurrency Restrictions of EACCEPTCOPY 39-98
Table 39-62. Additional Concurrency Restrictions of EACCEPTCOPY 39-98
Table 39-63. Base Concurrency Restrictions of EDECCSSA 39-101
Table 39-64. Additional Concurrency Restrictions of EDECCSSA 39-101
Table 39-65. Base Concurrency Restrictions of EENTER 39-106
Table 39-66. Additional Concurrency Restrictions of EENTER 39-106
Table 39-67. Base Concurrency Restrictions of EEXIT 39-114
Table 39-68. Additional Concurrency Restrictions of EEXIT 39-114
Table 39-69. Key Derivation 39-118
Table 39-70. EGETKEY Return Value in RAX 39-118
Table 39-71. Base Concurrency Restrictions of EGETKEY 39-118
Table 39-72. Additional Concurrency Restrictions of EGETKEY 39-119
Table 39-73. Base Concurrency Restrictions of EMODPE 39-127
Table 39-74. Additional Concurrency Restrictions of EMODPE 39-127
Table 39-75. Base Concurrency Restrictions of EREPORT 39-131
Table 39-76. Additional Concurrency Restrictions of EREPORT 39-131
Table 39-77. Base Concurrency Restrictions of ERESUME 39-136
Table 39-78. Additional Concurrency Restrictions of ERESUME 39-136
Table 39-79. Base Concurrency Restrictions of EDECVIRTCHILD 39-148
Table 39-80. Additional Concurrency Restrictions of EDECVIRTCHILD 39-149
Table 39-81. Base Concurrency Restrictions of EINCVIRTCHILD 39-152
Table 39-82. Additional Concurrency Restrictions of EINCVIRTCHILD 39-153
Table 39-83. Base Concurrency Restrictions of ESETCONTEXT 39-155
Table 39-84. Additional Concurrency Restrictions of ESETCONTEXT 39-156
Table 40-1. SGX Conflict Exit Qualification 40-4
Table 40-2. SMRAM Synthetic States on Asynchronous Enclave Exit 40-11
Table 40-3. Layout of the IA32_SGX_SVN_STATUS MSR 40-13
Table A-1. Memory Types Recommended for VMCS and Related Data Structures A-1
Table B-1. Encoding for 16-Bit Control Fields (0000_00xx_xxxx_xxx0B) B-1
Table B-2. Encodings for 16-Bit Guest-State Fields (0000_10xx_xxxx_xxx0B) B-1

PAGE

Table B-3. Encodings for 16-Bit Host-State Fields (0000_11xx_xxxx_xxx0B) B-2
Table B-4. Encodings for 64-Bit Control Fields (0010_00xx_xxxx_xxxAb) B-2
Table B-5. Encodings for 64-Bit Read-Only Data Fields (0010_01xx_xxxx_xxxAb) B-5
Table B-6. Encodings for 64-Bit Guest-State Fields (0010_10xx_xxxx_xxxAb) B-5
Table B-7. Encodings for 64-Bit Host-State Fields (0010_11xx_xxxx_xxxAb) B-7
Table B-8. Encodings for 32-Bit Control Fields (0100_00xx_xxxx_xxx0B) B-7
Table B-9. Encodings for 32-Bit Read-Only Data Fields (0100_01xx_xxxx_xxx0B) B-8
Table B-10. Encodings for 32-Bit Guest-State Fields (0100_10xx_xxxx_xxx0B) B-8
Table B-11. Encoding for 32-Bit Host-State Field (0100_11xx_xxxx_xxx0B) B-9
Table B-12. Encodings for Natural-Width Control Fields (0110_00xx_xxxx_xxx0B) B-10
Table B-13. Encodings for Natural-Width Read-Only Data Fields (0110_01xx_xxxx_xxx0B) B-10
Table B-14. Encodings for Natural-Width Guest-State Fields (0110_10xx_xxxx_xxx0B) B-10
Table B-15. Encodings for Natural-Width Host-State Fields (0110_11xx_xxxx_xxx0B) B-11
Table C-1. Basic Exit Reasons C-1

CHAPTER 1
ABOUT THIS MANUAL

The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1 (order number 253668), the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2 (order number 253669), the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3 (order number 326019), and the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:System Programming Guide, Part 4 (order number 332831) are part of a set that describes the architecture and programming environment of Intel 64 and IA-32 Architecture processors. The other volumes in this set are:

- Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture (order number 253665).
- Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D: Instruction Set Reference (order numbers 253666, 253667, 326018, and 334569).
- The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers (order number 335592).

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

1.1 OVERVIEW OF THE SYSTEM PROGRAMMING GUIDE

A description of this manual's content follows:

Chapter 1 — About This Manual. Gives an overview of all volumes of the Intel® 64 and IA-32 Architectures Software Developer's Manual, with chapter-specific details for the current volume.

Chapter 2 — System Architecture Overview. Describes the modes of operation used by Intel 64 and IA-32 processors and the mechanisms provided by the architectures to support operating systems and executives, including the system-oriented registers and data structures and the system-oriented instructions. The steps necessary for switching between real-address and protected modes are also identified.

Chapter 3 — Protected-Mode Memory Management. Describes the data structures, registers, and instructions that support segmentation and paging. The chapter explains how they can be used to implement a "flat" (unsegmented) memory model or a segmented memory model.

Chapter 4 — Linear-Address Pre-Processing. Describes the processes to which linear addresses are subject prior to translation changing. These include fault checking for linear-address-space separation (LASS) and canonicality, as well as linear-address masking (LAM).

Chapter 5 — Paging. Describes the paging modes supported by Intel 64 and IA-32 processors.

Chapter 6 — Protection. Describes the support for page and segment protection provided in the Intel 64 and IA-32 architectures. This chapter also explains the implementation of privilege rules, stack switching, pointer validation, user mode, and supervisor mode.

Chapter 7 — Interrupt and Exception Handling. Describes the basic interrupt mechanisms defined in the Intel 64 and IA-32 architectures, shows how interrupts and exceptions relate to protection, and describes how the architecture handles each exception type. Reference information for each exception is given in this chapter. Includes

programming the LINT0 and LINT1 inputs and gives an example of how to program the LINT0 and LINT1 pins for specific interrupt vectors.

Chapter 8 — User Interrupts. Describes user interrupts supported by Intel 64 and IA-32 processors.

Chapter 9 — Task Management. Describes mechanisms the Intel 64 and IA-32 architectures provide to support multitasking and inter-task protection.

Chapter 10 — Multiple-Processor Management. Describes the instructions and flags that support multiple processors with shared memory, memory ordering, and Intel® Hyper-Threading Technology. Includes MP initialization for P6 family processors and gives an example of how to use the MP protocol to boot P6 family processors in an MP system.

Chapter 11 — Processor Management and Initialization. Defines the state of an Intel 64 or IA-32 processor after reset initialization. This chapter also explains how to set up an Intel 64 or IA-32 processor for real-address mode operation and protected- mode operation, and how to switch between modes.

Chapter 12 — Advanced Programmable Interrupt Controller (APIC). Describes the programming interface to the local APIC and gives an overview of the interface between the local APIC and the I/O APIC. Includes APIC bus message formats and describes the message formats for messages transmitted on the APIC bus for P6 family and Pentium processors.

Chapter 13 — Memory Cache Control. Describes the general concept of caching and the caching mechanisms supported by the Intel 64 or IA-32 architectures. This chapter also describes the memory type range registers (MTRRs) and how they can be used to map memory types of physical memory. Information on using the new cache control and memory streaming instructions introduced with the Pentium III, Pentium 4, and Intel Xeon processors is also given.

Chapter 14 — Intel® MMX™ Technology System Programming. Describes those aspects of the Intel® MMX™ technology that must be handled and considered at the system programming level, including: task switching, exception handling, and compatibility with existing system environments.

Chapter 15 — System Programming For Instruction Set Extensions And Processor Extended States. Describes the operating system requirements to support SSE/SSE2/SSE3/SSSE3/SSE4 extensions, including task switching, exception handling, and compatibility with existing system environments. The latter part of this chapter describes the extensible framework of operating system requirements to support processor extended states. Processor extended state may be required by instruction set extensions beyond those of SSE/SSE2/SSE3/SSSE3/SSE4 extensions.

Chapter 16 — Power and Thermal Management. Describes facilities of Intel 64 and IA-32 architecture used for power management and thermal monitoring.

Chapter 17 — Machine-Check Architecture. Describes the machine-check architecture and machine-check exception mechanism found in the Pentium 4, Intel Xeon, and P6 family processors. Additionally, a signaling mechanism for software to respond to hardware corrected machine check error is covered.

Chapter 18 — Interpreting Machine-Check Error Codes. Gives an example of how to interpret the error codes for a machine-check error that occurred on a P6 family processor.

Chapter 19 — Debug, Branch Profile, TSC, and Resource Monitoring Features. Describes the debugging registers and other debug mechanism provided in Intel 64 or IA-32 processors. This chapter also describes the time-stamp counter.

Chapter 20 — Last Branch Records. Describes the Last Branch Records (architectural feature).

Chapter 21 — Performance Monitoring. Describes the Intel 64 and IA-32 architectures' facilities for monitoring performance.

Chapter 22 — 8086 Emulation. Describes the real-address and virtual-8086 modes of the IA-32 architecture.

Chapter 23 — Mixing 16-Bit and 32-Bit Code. Describes how to mix 16-bit and 32-bit code modules within the same program or task.

Chapter 24 — IA-32 Architecture Compatibility. Describes architectural compatibility among IA-32 processors.

Chapter 25 — Introduction to Virtual Machine Extensions. Describes the basic elements of virtual machine architecture and the virtual machine extensions for Intel 64 and IA-32 Architectures.

Chapter 26 — Virtual Machine Control Structures. Describes components that manage VMX operation. These include the working-VMCS pointer and the controlling-VMCS pointer.

Chapter 27 — VMX Non-Root Operation. Describes the operation of a VMX non-root operation. Processor operation in VMX non-root mode can be restricted programmatically such that certain operations, events or conditions can cause the processor to transfer control from the guest (running in VMX non-root mode) to the monitor software (running in VMX root mode).

Chapter 28 — VM Entries. Describes VM entries. VM entry transitions the processor from the VMM running in VMX root-mode to a VM running in VMX non-root mode. VM-Entry is performed by the execution of VMLAUNCH or VMRESUME instructions.

Chapter 29 — VM Exits. Describes VM exits. Certain events, operations or situations while the processor is in VMX non-root operation may cause VM-exit transitions. In addition, VM exits can also occur on failed VM entries.

Chapter 30 — VMX Support for Address Translation. Describes virtual-machine extensions that support address translation and the virtualization of physical memory.

Chapter 31 — APIC Virtualization and Virtual Interrupts. Describes the VMCS including controls that enable the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC).

Chapter 32 — VMX Instruction Reference. Describes the virtual-machine extensions (VMX). VMX is intended for a system executive to support virtualization of processor hardware and a system software layer acting as a host to multiple guest software environments.

Chapter 33 — System Management Mode. Describes Intel 64 and IA-32 architectures' system management mode (SMM) facilities.

Chapter 34 — Intel® Processor Trace. Describes details of Intel® Processor Trace.

Chapter 35 — Introduction to Intel® Software Guard Extensions. Provides an overview of the Intel® Software Guard Extensions (Intel® SGX) set of instructions.

Chapter 36 — Enclave Access Control and Data Structures. Describes Enclave Access Control procedures and defines various Intel SGX data structures.

Chapter 37 — Enclave Operation. Describes enclave creation and initialization, adding pages and measuring an enclave, and enclave entry and exit.

Chapter 38 — Enclave Exiting Events. Describes enclave-exiting events (EEE) and asynchronous enclave exit (AEX).

Chapter 39 — SGX Instruction References. Describes the supervisor and user level instructions provided by Intel SGX.

Chapter 40 — Intel® SGX Interactions with IA32 and Intel® 64 Architecture. Describes the Intel SGX collection of enclave instructions for creating protected execution environments on processors supporting IA32 and Intel 64 architectures.

Chapter 41 — Enclave Code Debug and Profiling. Describes enclave code debug processes and options.

Appendix A — VMX Capability Reporting Facility. Describes the VMX capability MSRs. Support for specific VMX features is determined by reading capability MSRs.

Appendix B — Field Encoding in VMCS. Enumerates all fields in the VMCS and their encodings. Fields are grouped by width (16-bit, 32-bit, etc.) and type (guest-state, host-state, etc.).

Appendix C — VM Basic Exit Reasons. Describes the 32-bit fields that encode reasons for a VM exit. Examples of exit reasons include, but are not limited to: software interrupts, processor exceptions, software traps, NMIs, external interrupts, and triple faults.

CHAPTER 2
SYSTEM ARCHITECTURE OVERVIEW

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

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

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

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

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

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

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

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

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

All Intel 64 and IA-32 processors enter real-address mode following a power-up or reset (see Chapter 11, "Processor Management and Initialization"). Software then initiates the switch from real-address mode to protected mode. If IA-32e mode operation is desired, software also initiates a switch from protected mode to IA-32e mode.

2.1 OVERVIEW OF THE SYSTEM-LEVEL ARCHITECTURE

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

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

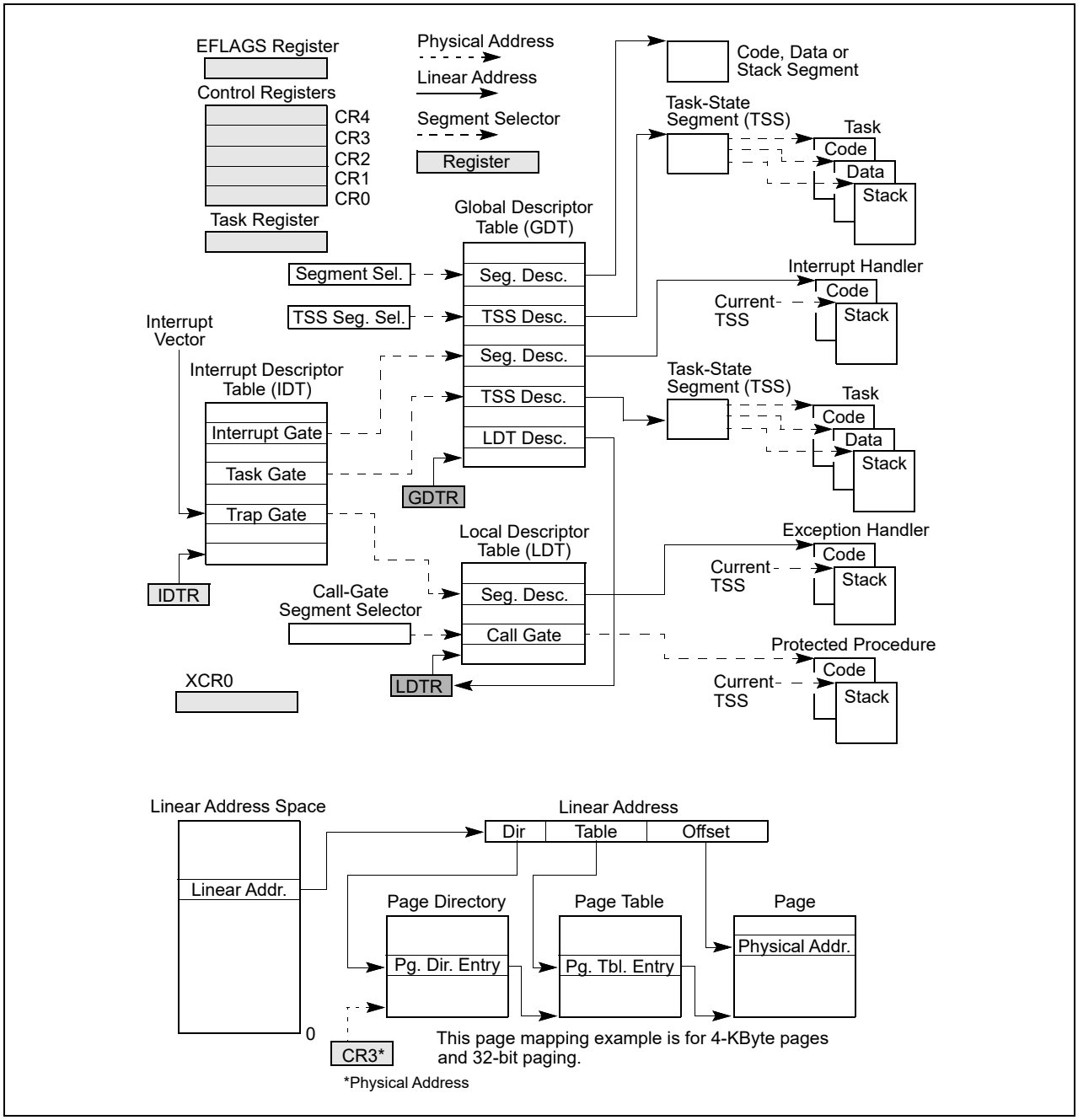


Figure 2-1. IA-32 System-Level Registers and Data Structures

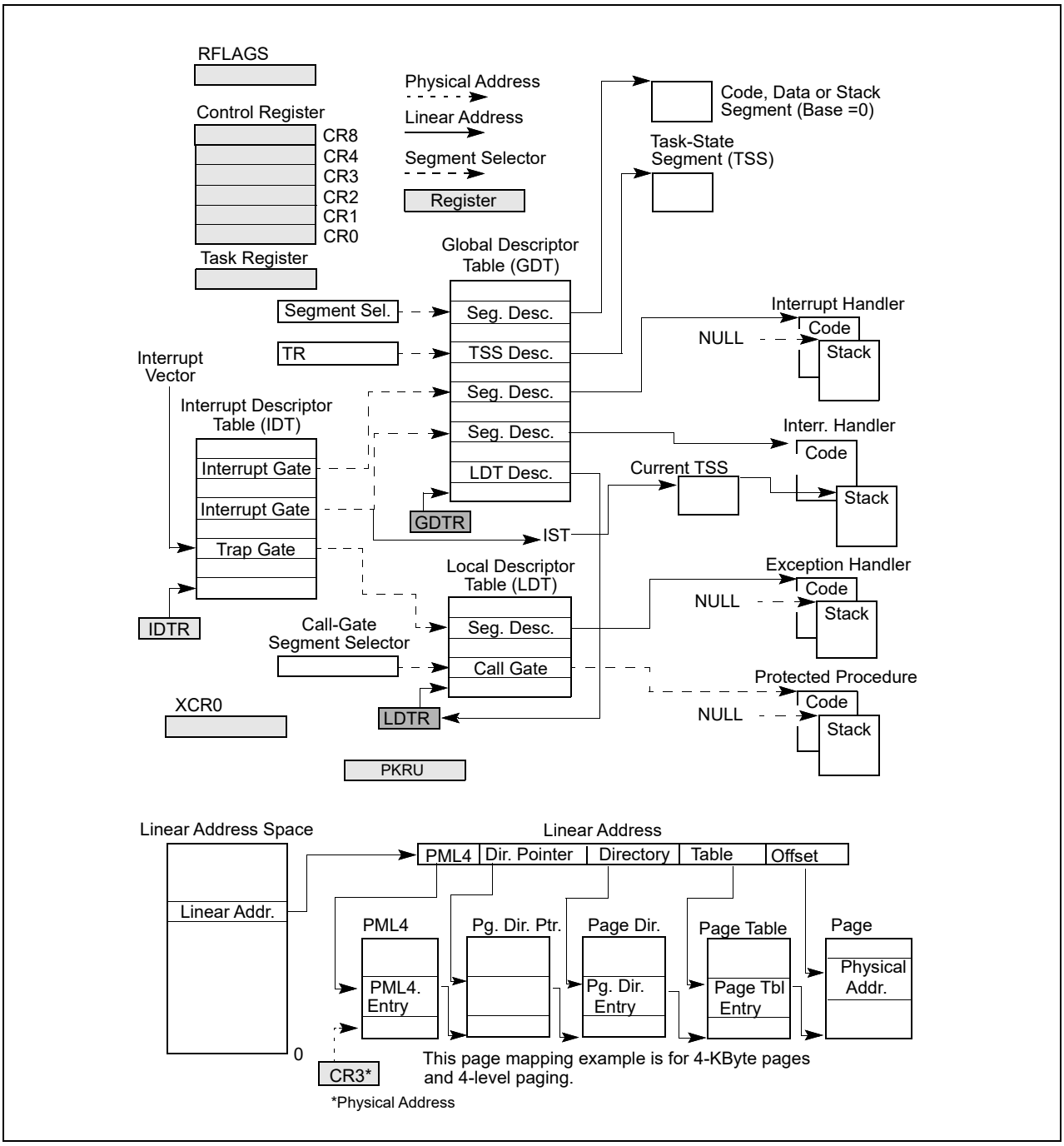


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode and 4-Level Paging

2.1.1 Global and Local Descriptor Tables

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

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

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

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

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

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

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

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

2.1.2 System Segments, Segment Descriptors, and Gates

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

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

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

2.1.2.1 Gates in IA-32e Mode

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

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

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

2.1.3 Task-State Segments and Task Gates

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

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

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

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

2.1.3.1 Task-State Segments in IA-32e Mode

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

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

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

The task register is expanded to hold 64-bit base addresses in IA-32e mode. See also: Section 9.7, "Task Management in 64-bit Mode."

2.1.4 Interrupt and Exception Handling

External interrupts, software interrupts and exceptions are handled through the interrupt descriptor table (IDT). The IDT stores a collection of gate descriptors that provide access to interrupt and exception handlers. Like the GDT, the IDT is not a segment. The linear address for the base of the IDT is contained in the IDT register (IDTR).

Gate descriptors in the IDT can be interrupt, trap, or task gate descriptors. To access an interrupt or exception handler, the processor first receives an interrupt vector from internal hardware, an external interrupt controller, or from software by means of an INT *n*, INTO, INT3, INT1, or BOUND instruction. The interrupt vector provides an index into the IDT. If the selected gate descriptor is an interrupt gate or a trap gate, the associated handler procedure is accessed in a manner similar to calling a procedure through a call gate. If the descriptor is a task gate, the handler is accessed through a task switch.

2.1.4.1 Interrupt and Exception Handling IA-32e Mode

In IA-32e mode, interrupt gate descriptors are expanded to 16 bytes to support 64-bit base addresses. This is true for 64-bit mode and compatibility mode.

The IDTR register is expanded to hold a 64-bit base address. Task gates are not supported.

2.1.5 Memory Management

System architecture supports either direct physical addressing of memory or virtual memory (through paging). When physical addressing is used, a linear address is treated as a physical address. When paging is used: all code, data, stack, and system segments (including the GDT and IDT) can be paged with only the most recently accessed pages being held in physical memory.

The location of pages (sometimes called page frames) in physical memory is contained in the paging structures. These structures reside in physical memory (see Figure 2-1 for the case of 32-bit paging).

The base physical address of the paging-structure hierarchy is contained in control register CR3. The entries in the paging structures determine the physical address of the base of a page frame, access rights and memory management information.

To use this paging mechanism, a linear address is broken into parts. The parts provide separate offsets into the paging structures and the page frame. A system can have a single hierarchy of paging structures or several. For example, each task can have its own hierarchy.

2.1.5.1 Memory Management in IA-32e Mode

In IA-32e mode, physical memory pages are managed by a set of system data structures. In both compatibility mode and 64-bit mode, four or five levels of system data structures are used (see Chapter 5, "Paging"). These include the following:

- **The page map level 5 (PML5)** — An entry in the PML5 table contains the physical address of the base of a PML4 table, access rights, and memory management information. The base physical address of the PML5 table is stored in CR3. The PML5 table is used only with 5-level paging.
- **A page map level 4 (PML4)** — An entry in a PML4 table contains the physical address of the base of a page directory pointer table, access rights, and memory management information. With 4-level paging, there is only one PML4 table and its base physical address is stored in CR3.
- **A set of page directory pointer tables** — An entry in a page directory pointer table contains the physical address of the base of a page directory table, access rights, and memory management information.
- **Sets of page directories** — An entry in a page directory table contains the physical address of the base of a page table, access rights, and memory management information.
- **Sets of page tables** — An entry in a page table contains the physical address of a page frame, access rights, and memory management information.

2.1.6 System Registers

To assist in initializing the processor and controlling system operations, the system architecture provides system flags in the EFLAGS register and several system registers:

- The system flags and IOPL field in the EFLAGS register control task and mode switching, interrupt handling, instruction tracing, and access rights. See also: Section 2.3, "System Flags and Fields in the EFLAGS Register."
- The control registers (CR0, CR2, CR3, and CR4) contain a variety of flags and data fields for controlling system-level operations. Other flags in these registers are used to indicate support for specific processor capabilities within the operating system or executive. See also: Chapter 2, "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 19, "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 11.4, "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.

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 13, "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.

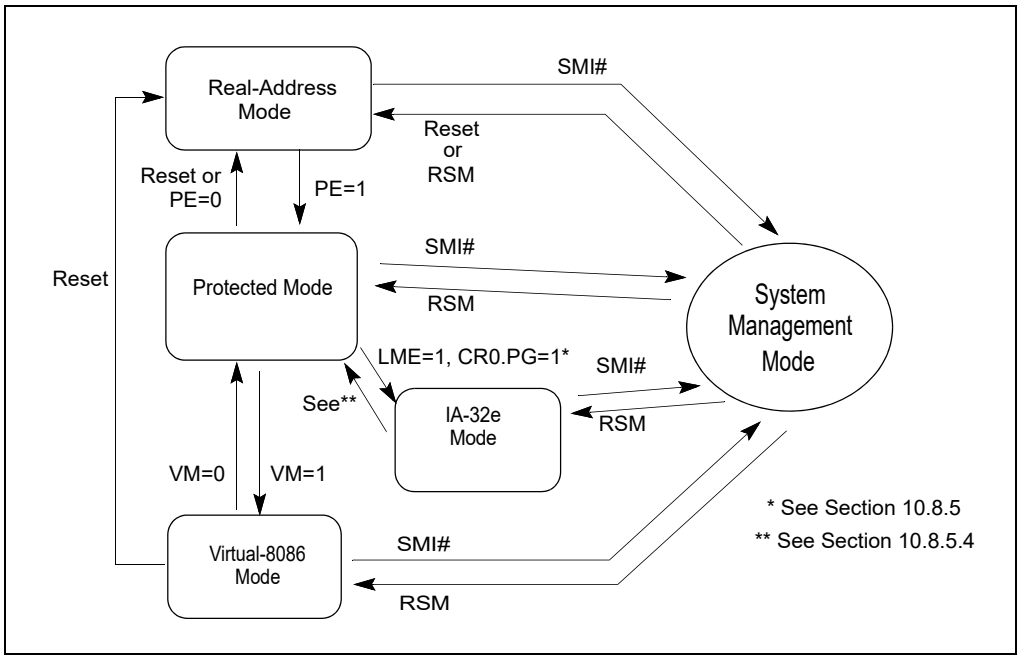


Figure 2-3. Transitions Among the Processor's Operating Modes

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 11.9, "Mode Switching," and Section 5.1.2, "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 22.2.5, "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 11, "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 5.6, “Access Rights”). The layout of the IA32_EFER MSR is shown in Figure 2-4.

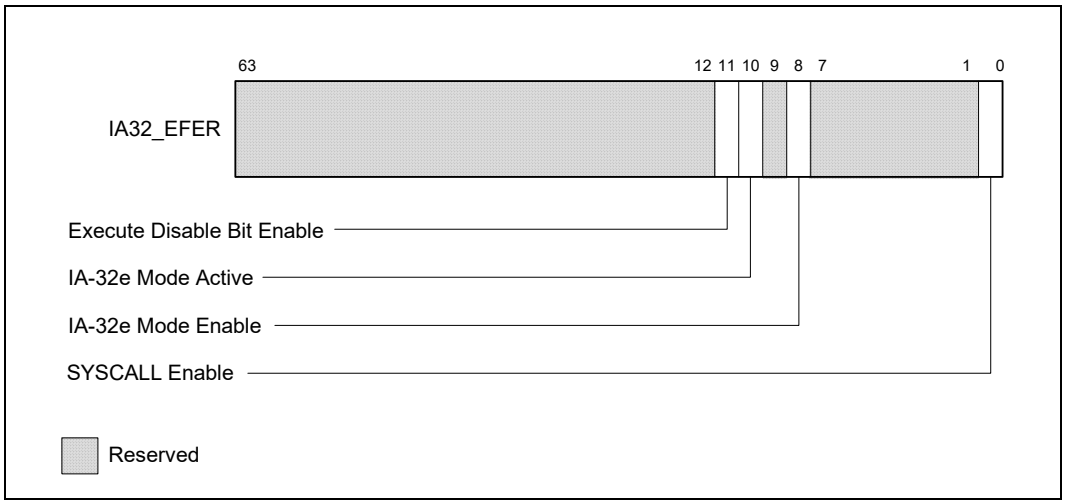


Figure 2-4. IA32_EFER MSR Layout

Table 2-1. IA32_EFER MSR Information

Bit	Description
0	**SYSCALL Enable: IA32_EFER.SCE (R/W)** Enables SYSCALL/SYSRET instructions in 64-bit mode.
7:1	Reserved.
8	**IA-32e Mode Enable: IA32_EFER.LME (R/W)** Enables IA-32e mode operation.
9	Reserved.
10	**IA-32e Mode Active: IA32_EFER.LMA (R)** Indicates IA-32e mode is active when set.
11	**Execute Disable Bit Enable: IA32_EFER.NXE (R/W)** Enables page access restriction by preventing instruction fetches from PAE pages with the XD bit set (See Section 5.6).
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.

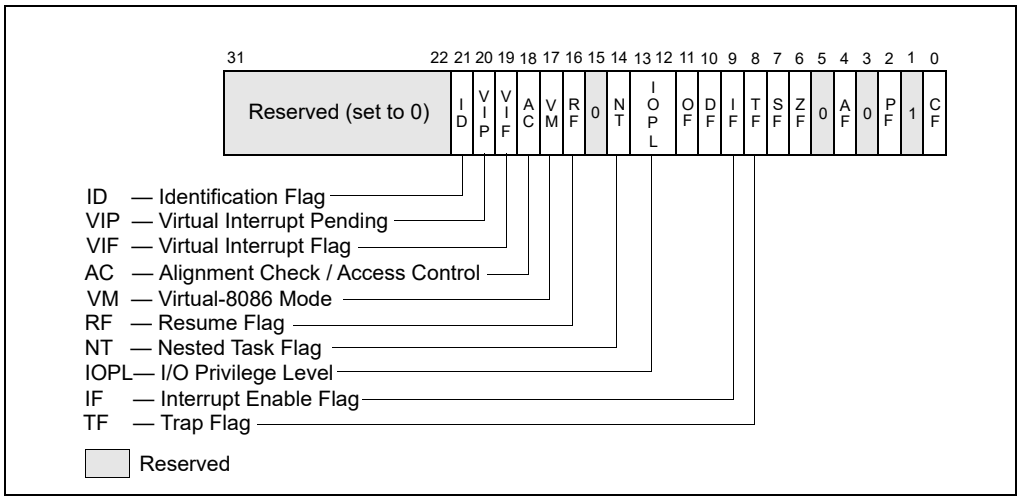


Figure 2-5. System Flags in the EFLAGS Register

IF **Interrupt enable (bit 9)** — Controls the response of the processor to maskable hardware interrupt requests (see also: Section 7.3.2, "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 20, "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 9.4, "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 19.3.1.1, "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 22.2.1, "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 5.6, "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 22.3.3.5, "Method 6: Software Interrupt Handling," and Section 22.4, "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 22.3.3.5, "Method 6: Software Interrupt Handling," and Section 22.4, "Protected-Mode Virtual Interrupts."

ID **Identification (bit 21)** — The ability of a program or procedure to set or clear this flag indicates support for the CPUID instruction.

2.3.1 System Flags and Fields in IA-32e Mode

In 64-bit mode, the RFLAGS register expands to 64 bits with the upper 32 bits reserved. System flags in RFLAGS (64-bit mode) or EFLAGS (compatibility mode) are shown in Figure 2-5.

In IA-32e mode, the processor does not allow the VM bit to be set because virtual-8086 mode is not supported (attempts to set the bit are ignored). Also, the processor will not set the NT bit. The processor does, however, allow software to set the NT bit (note that an IRET causes a general protection fault in IA-32e mode if the NT bit is set).

In IA-32e mode, the SYSCALL/SYSRET instructions have a programmable method of specifying which bits are cleared in RFLAGS/EFLAGS. These instructions save/restore EFLAGS/RFLAGS.

2.4 MEMORY-MANAGEMENT REGISTERS

The processor provides four memory-management registers (GDTR, LDTR, IDTR, and TR) that specify the locations of the data structures which control segmented memory management (see Figure 2-6). Special instructions are provided for loading and storing these registers.

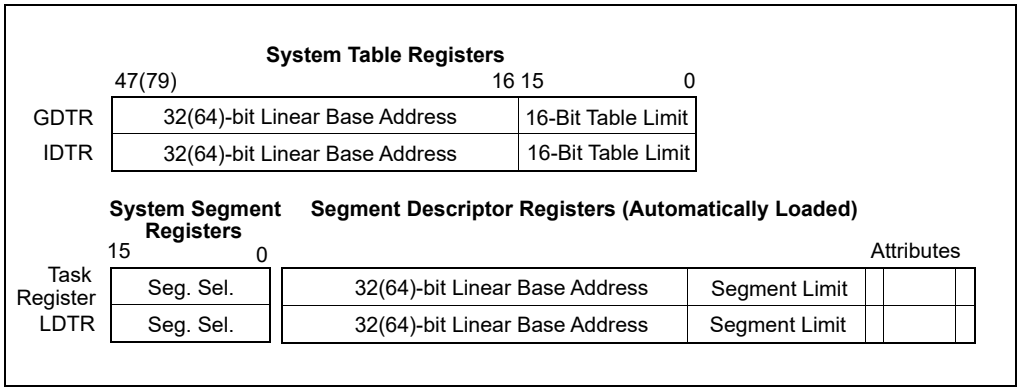


Figure 2-6. Memory Management Registers

2.4.1 Global Descriptor Table Register (GDTR)

The GDTR register holds the base address (32 bits in protected mode; 64 bits in IA-32e mode) and the 16-bit table limit for the GDT. The base address specifies the linear address of byte 0 of the GDT; the table limit specifies the number of bytes in the table.

The LGDT and SGDT instructions load and store the GDTR register, respectively. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH. A new base address must be loaded into the GDTR as part of the processor initialization process for protected-mode operation.

See also: Section 3.5.1, "Segment Descriptor Tables."

2.4.2 Local Descriptor Table Register (LDTR)

The LDTR register holds the 16-bit segment selector, base address (32 bits in protected mode; 64 bits in IA-32e mode), segment limit, and descriptor attributes for the LDT. The base address specifies the linear address of byte 0 of the LDT segment; the segment limit specifies the number of bytes in the segment. See also: Section 3.5.1, "Segment Descriptor Tables."

The LLDT and SLDT instructions load and store the segment selector part of the LDTR register, respectively. The segment that contains the LDT must have a segment descriptor in the GDT. When the LLDT instruction loads a segment selector in the LDTR: the base address, limit, and descriptor attributes from the LDT descriptor are automatically loaded in the LDTR.

When a task switch occurs, the LDTR is automatically loaded with the segment selector and descriptor for the LDT for the new task. The contents of the LDTR are not automatically saved prior to writing the new LDT information into the register.

On power up or reset of the processor, the segment selector and base address are set to the default value of 0 and the limit is set to 0FFFFH.

2.4.3 IDTR Interrupt Descriptor Table Register

The IDTR register holds the base address (32 bits in protected mode; 64 bits in IA-32e mode) and 16-bit table limit for the IDT. The base address specifies the linear address of byte 0 of the IDT; the table limit specifies the number of bytes in the table. The LIDT and SIDT instructions load and store the IDTR register, respectively. On power up or reset of the processor, the base address is set to the default value of 0 and the limit is set to 0FFFFH. The base address and limit in the register can then be changed as part of the processor initialization process.

See also: Section 7.10, "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 9.2.4, "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.
- Some of the bits in CR0 and CR4 are reserved and must be written with zeros. Attempting to set any reserved bits in CR0[31:0] is ignored. Attempting to set any reserved bits in CR0[63:32] results in a general-protection exception, #GP(0). Attempting to set any reserved bits in CR4 results in a general-protection exception, #GP(0).
- All 64 bits of CR2 are writable by software.
- Bits in CR3 in the range 63:MAXPHYADDR that are reserved (see Figure 2-7) must be zero. Attempting to set any of them results in #GP(0).
- The MOV CR2 instruction does not check that address written to CR2 is canonical.
- A 64-bit capable processor will retain the upper 32 bits of each control register when transitioning out of IA-32e mode.
- On a 64-bit capable processor, an execution of MOV to CR outside of 64-bit mode zeros the upper 32 bits of the control register.
- Register CR8 is available in 64-bit mode only.

The control registers are summarized below, and each architecturally defined control field in these control registers is described individually. In Figure 2-7, the width of the register in 64-bit mode is indicated in parenthesis (except for CR0).

- **CR0** — Contains system control flags that control operating mode and states of the processor.
- **CR1** — Reserved.
- **CR2** — Contains the page-fault linear address (the linear address that caused a page fault).
- **CR3** — Contains the physical address of the base of the paging-structure hierarchy and two flags (PCD and PWT). Only the most-significant bits (less the lower 12 bits) of the base address are specified; the lower 12 bits of the address are assumed to be 0. The first paging structure must thus be aligned to a page (4-KByte) boundary. The PCD and PWT flags control caching of that paging structure in the processor's internal data caches (they do not control TLB caching of page-directory information).

  When using the physical address extension, the CR3 register contains the base address of the page-directory-pointer table. With 4-level paging and 5-level paging, the CR3 register contains the base address of the PML4

table and PML5 table, respectively. If PCIDs are enabled, CR3 has a format different from that illustrated in Figure 2-7. See Section 5.5, "4-Level Paging and 5-Level Paging."

When linear-address masking is supported, CR3 includes two bits that control the masking of user pointers (see Section 4.4, "Linear-Address Masking."

See also: Chapter 5, "Paging."

- **CR4** — Contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific processor capabilities. Bits CR4[63:32] can only be used for IA-32e mode only features that are enabled after entering 64-bit mode. Bits CR4[63:32] do not have any effect outside of IA-32e mode.
- **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.

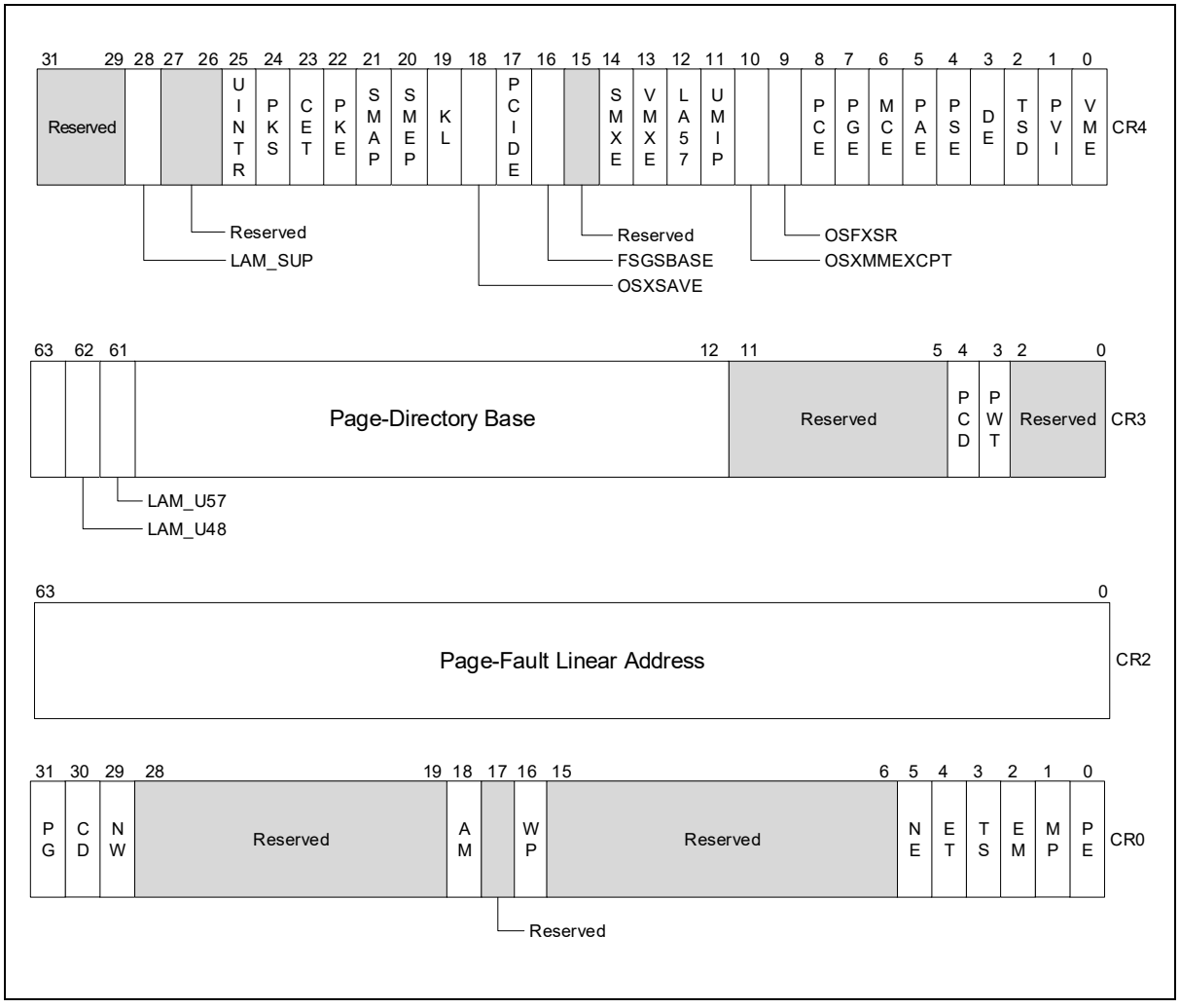


Figure 2-7. Control Registers

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 5, "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 13-5. 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 13.5.3, "Preventing Caching," and Section 13.5, "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 13-5 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 5.1.3 and Section 5.6). This flag facilitates implementation of the copy-on-write method of creating a new process (forking) used by operating systems such as UNIX. This flag must be set before software can set CR4.CET, and it cannot be cleared as long as CR4.CET = 1 (see below).

CR0.NE

Numeric Error (bit 5 of CR0) — Enables the native (internal) mechanism for reporting x87 FPU errors when set; enables the PC-style x87 FPU error reporting mechanism when clear. When the NE flag is clear and the IGNNE# input is asserted, x87 FPU errors are ignored. When the NE flag is clear and the IGNNE# input is deasserted, an unmasked x87 FPU error causes the processor to assert the FERR# pin to generate an external interrupt and to stop instruction execution immediately before executing the next waiting floating-point instruction or WAIT/FWAIT instruction.

The FERR# pin is intended to drive an input to an external interrupt controller (the FERR# pin emulates the ERROR# pin of the Intel 287 and Intel 387 DX math coprocessors). The NE flag, IGNNE# pin, and FERR# pin are used with external logic to implement PC-style error reporting. Using FERR# and IGNNE# to handle floating-point exceptions is deprecated by modern operating systems; this non-native approach also limits newer processors to operate with one logical processor active.

See also: Section 8.7, "Handling x87 FPU Exceptions in Software," in Chapter 8, "Programming with the x87 FPU," and Appendix A, "EFLAGS Cross-Reference," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

CR0.ET

Extension Type (bit 4 of CR0) — Reserved in the Pentium 4, Intel Xeon, P6 family, and Pentium processors. In the Pentium 4, Intel Xeon, and P6 family processors, this flag is hardcoded to 1. In the Intel386 and Intel486 processors, this flag indicates support of Intel 387 DX math coprocessor instructions when set.

CR0.TS

Task Switched (bit 3 of CR0) — Allows the saving of the x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 context on a task switch to be delayed until an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction is

actually executed by the new task. The processor sets this flag on every task switch and tests it when executing x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

- If the TS flag is set and the EM flag (bit 2 of CR0) is clear, a device-not-available exception (#NM) is raised prior to the execution of any x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction; with the exception of PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT. See the paragraph below for the special case of the WAIT/FWAIT instructions.
- If the TS flag is set and the MP flag (bit 1 of CR0) and EM flag are clear, an #NM exception is not raised prior to the execution of an x87 FPU WAIT/FWAIT instruction.
- If the EM flag is set, the setting of the TS flag has no effect on the execution of x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

Table 2-2 shows the actions taken when the processor encounters an x87 FPU instruction based on the settings of the TS, EM, and MP flags. Table 14-1 and 15-1 show the actions taken when the processor encounters an MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction.

The processor does not automatically save the context of the x87 FPU, XMM, and MXCSR registers on a task switch. Instead, it sets the TS flag, which causes the processor to raise an #NM exception whenever it encounters an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction in the instruction stream for the new task (with the exception of the instructions listed above).

The fault handler for the #NM exception can then be used to clear the TS flag (with the CLTS instruction) and save the context of the x87 FPU, XMM, and MXCSR registers. If the task never encounters an x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instruction, the x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3/SSE4 context is never saved.

Table 2-2. Action Taken By x87 FPU Instructions for Different Combinations of EM, MP, and TS

CR0 Flags			x87 FPU Instruction Type	
EM	MP	TS	Floating-Point	WAIT/FWAIT
0	0	0	Execute	Execute.
0	0	1	#NM Exception	Execute.
0	1	0	Execute	Execute.
0	1	1	#NM Exception	#NM exception.
1	0	0	#NM Exception	Execute.
1	0	1	#NM Exception	Execute.
1	1	0	#NM Exception	Execute.
1	1	1	#NM Exception	#NM exception.

CR0.EM

Emulation (bit 2 of CR0) — Indicates that the processor does not have an internal or external x87 FPU when set; indicates an x87 FPU is present when clear. This flag also affects the execution of MMX/SSE/SSE2/SSE3/SSSE3/SSE4 instructions.

When the EM flag is set, execution of an x87 FPU instruction generates a device-not-available exception (#NM). This flag must be set when the processor does not have an internal x87 FPU or is not connected to an external math coprocessor. Setting this flag forces all floating-point instructions to be handled by software emulation. Table 11-3 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 14-1). 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 15-1). 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 CR0) — Controls the interaction of the WAIT (or FWAIT) instruction with the TS flag (bit 3 of CR0). If the MP flag is set, a WAIT instruction generates a device-not-available exception (#NM) if the TS flag is also set. If the MP flag is clear, the WAIT instruction ignores the setting of the TS flag. Table 11-3 shows the recommended setting of this flag, depending on the IA-32 processor and x87 FPU or math coprocessor present in the system. Table 2-2 shows the interaction of the MP, EM, and TS flags.

CR0.PE

Protection Enable (bit 0 of CR0) — Enables protected mode when set; enables real-address mode when clear. This flag does not enable paging directly. It only enables segment-level protection. To enable paging, both the PE and PG flags must be set.

See also: Section 11.9, "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 5.9, "Paging and Memory Typing." This bit is not used if paging is disabled, with PAE paging, or with 4-level paging[1] or 5-level paging if CR4.PCIDE=1.

CR3.PWT

Page-level Write-Through (bit 3 of CR3) — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. See Section 5.9, "Paging and Memory Typing." This bit is not used if paging is disabled, with PAE paging, or with 4-level paging or 5-level paging if CR4.PCIDE=1.

CR3.LAM_U57

User LAM57 enable (bit 61 of CR3) — When set, enables LAM57 (masking of linear-address bits 62:57) for user pointers and overrides CR3.LAM_U48. See Section 4.4, "Linear-Address Masking."

CR3.LAM_U48

User LAM48 enable (bit 62 of CR3) — When set and CR3.LAM_U57 is clear, enables LAM48 (masking of linear-address bits 62:48) for user pointers. See Section 4.4, "Linear-Address Masking."

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 22.3, "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 22.4, "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.

---

1. Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.

See also: Section 19.2.2, "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 5.3, "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 5, "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 17, "Machine-Check Architecture."

CR4.PGE

Page Global Enable (bit 7 of CR4) — (Introduced in the P6 family processors.) Enables the global page feature when set; disables the global page feature when clear. The global page feature allows frequently used or shared pages to be marked as global to all users (done with the global flag, bit 8, in a page-directory-pointer-table entry, a page-directory entry, or a page-table entry). Global pages are not flushed from the translation-lookaside buffer (TLB) on a task switch or a write to register CR3.

When enabling the global page feature, paging must be enabled (by setting the PG flag in control register CR0) before the PGE flag is set. Reversing this sequence may affect program correctness, and processor performance will be impacted.

See also: Section 5.10, "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, PREFETCH*h*, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT.

If this flag is clear, the FXSAVE and FXRSTOR instructions will save and restore the contents of the x87 FPU and MMX registers, but they may not save and restore the contents of the XMM and MXCSR registers. Also, the processor will generate an invalid opcode exception (#UD) if it attempts to execute any SSE/SSE2/SSE3 instruction, with the exception of PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, MOVNTI, CLFLUSH, CRC32, and POPCNT. The operating system or executive must explicitly set this flag.

NOTE

CPUID feature flag FXSR indicates availability of the FXSAVE/FXRSTOR instructions. The OSFXSR bit provides operating system software with a means of enabling FXSAVE/FXRSTOR to save/restore the contents of the X87 FPU, XMM, and MXCSR registers. Consequently OSFXSR bit indicates that the operating system provides context switch support for SSE/SSE2/SSE3/SSSE3/SSE4.

CR4.OSXMMEXCPT

Operating System Support for Unmasked SIMD Floating-Point Exceptions (bit 10 of CR4) — When set, indicates that the operating system supports the handling of unmasked SIMD floating-point exceptions through an exception handler that is invoked when a SIMD floating-point exception (#XM) is

generated. SIMD floating-point exceptions are only generated by SSE/SSE2/SSE3/SSE4.1 SIMD floating-point instructions.

The operating system or executive must explicitly set this flag. If this flag is not set, the processor will generate an invalid opcode exception (#UD) whenever it detects an unmasked SIMD floating-point exception.

CR4.UMIP

User-Mode Instruction Prevention (bit 11 of CR4) — When set, the following instructions cannot be executed if CPL > 0: SGDT, SIDT, SLDT, SMSW, and STR. An attempt at such execution causes a general-protection exception (#GP).

CR4.LA57

57-bit linear addresses (bit 12 of CR4) — When set in IA-32e mode, the processor uses 5-level paging to translate 57-bit linear addresses. When clear in IA-32e mode, the processor uses 4-level paging to translate 48-bit linear addresses. This bit cannot be modified in IA-32e mode.

See also: Chapter 5, "Paging."

CR4.VMXE

VMX-Enable Bit (bit 13 of CR4) — Enables VMX operation when set. See Chapter 25, "Introduction to Virtual Machine Extensions."

CR4.SMXE

SMX-Enable Bit (bit 14 of CR4) — Enables SMX operation when set. See Chapter 7, "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.PCIDE

PCID-Enable Bit (bit 17 of CR4) — Enables process-context identifiers (PCIDs) when set. See Section 5.10.1, "Process-Context Identifiers (PCIDs)." Applies only in IA-32e mode (if IA32_EFER.LMA = 1).

CR4.OSXSAVE

XSAVE and Processor Extended States-Enable Bit (bit 18 of CR4) — When set, this flag: (1) indicates (via CPUID.01H:ECX.OSXSAVE[bit 27]) that the operating system supports the use of the XGETBV, XSAVE, and XRSTOR instructions by general software; (2) enables the XSAVE and XRSTOR instructions to save and restore the x87 FPU state (including MMX registers), the SSE state (XMM registers and MXCSR), along with other processor extended states enabled in XCR0; (3) enables the processor to execute XGETBV and XSETBV instructions in order to read and write XCR0. See Section 2.6 and Chapter 15, "System Programming for Instruction Set Extensions and Processor Extended States."

CR4.KL

Key-Locker-Enable Bit (bit 19 of CR4) — When set, the LOADIWKEY instruction is enabled; in addition, if support for the AES Key Locker instructions has been activated by system firmware, CPUID.19H:EBX.AESKLE[bit 0] is enumerated as 1 and the AES Key Locker instructions are enabled.[1] When clear, CPUID.19H:EBX.AESKLE[bit 0] is enumerated as 0 and execution of any Key Locker instruction causes an invalid-opcode exception (#UD).

CR4.SMEP

SMEP-Enable Bit (bit 20 of CR4) — Enables supervisor-mode execution prevention (SMEP) when set. See Section 5.6, "Access Rights."

CR4.SMAP

SMAP-Enable Bit (bit 21 of CR4) — Enables supervisor-mode access prevention (SMAP) when set. See Section 5.6, "Access Rights."

CR4.PKE

Enable protection keys for user-mode pages (bit 22 of CR4) — 4-level paging and 5-level paging

1. Software can check CPUID.19H:EBX.AESKLE[bit 0] after setting CR4.KL to determine whether the AES Key Locker instructions have been enabled. Note that some processors may allow enabling of those instructions without activation by system firmware. Some processors may not support use of the AES Key Locker instructions in system-management mode (SMM). Those processors enumerate CPUID.19H:EBX.AESKLE[bit 0] as 0 in SMM regardless of the setting of CR4.KL.

associate each user-mode linear address with a protection key. When set, this flag indicates (via CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4]) that the operating system supports use of the PKRU register to specify, for each protection key, whether user-mode linear addresses with that protection key can be read or written. This bit also enables access to the PKRU register using the RDPKRU and WRPKRU instructions.

CR4.CET
: **Control-flow Enforcement Technology (bit 23 of CR4)** — Enables control-flow enforcement technology when set. See Chapter 18, "Control-flow Enforcement Technology (CET)," of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 1*. This flag can be set only if CR0.WP is set, and it must be clear before CR0.WP can be cleared (see below).

CR4.PKS
: **Enable protection keys for supervisor-mode pages (bit 24 of CR4)** — 4-level paging and 5-level paging associate each supervisor-mode linear address with a protection key. When set, this flag allows use of the IA32_PKRS MSR to specify, for each protection key, whether supervisor-mode linear addresses with that protection key can be read or written.

CR4.UINTR
: **User Interrupts Enable Bit (bit 25 of CR4)** — Enables user interrupts when set, including user-interrupt delivery, user-interrupt notification identification, and the user-interrupt instructions.

CR4.LAM_SUP
: **Supervisor LAM enable (bit 28 of CR4)** — When set, enables LAM (linear-address masking) for supervisor pointers. See Section 4.4, "Linear-Address Masking."

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.

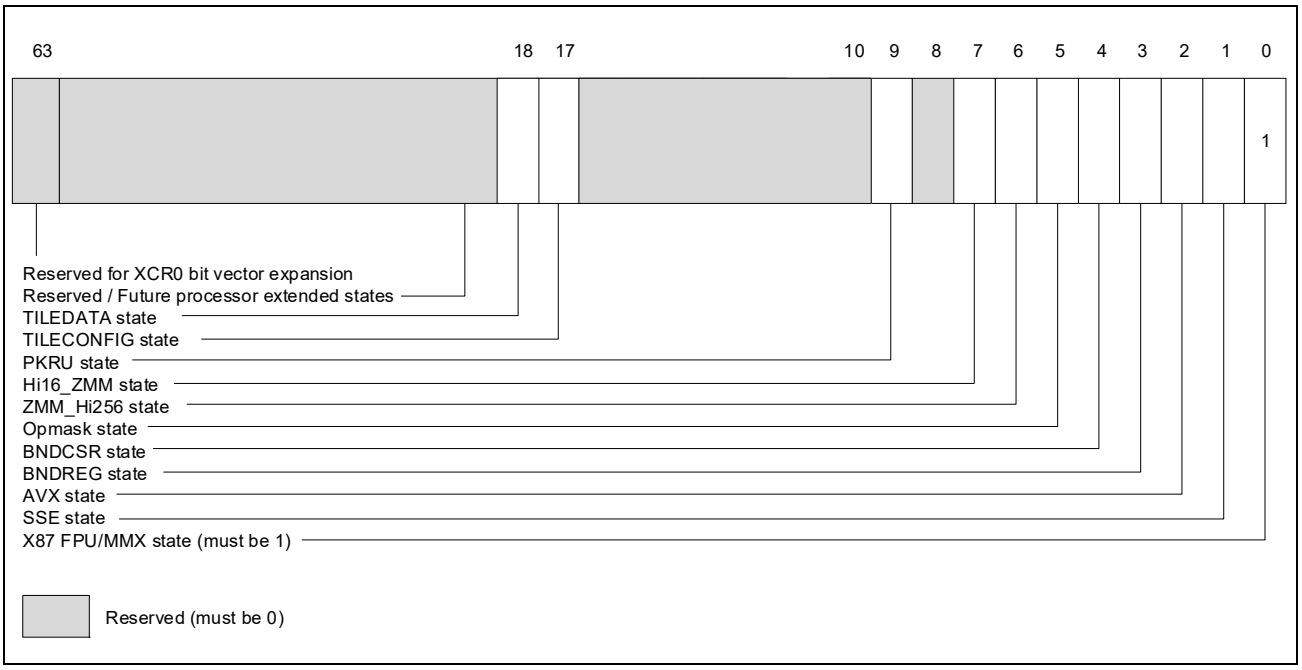


Figure 2-8. XCR0

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, Intel 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, Intel 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, Intel 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, Intel 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, Intel 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, Intel 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).
- XCR0.TILECFG (bit 17): If 1, and if XCR0.TILEDATA is also 1, Intel AMX instructions can be executed and the XSAVE feature set can be used to manage TILECFG.

- XCR0.TILEDATA (bit 18): If 1, and if XCR0.TILECFG is also 1, Intel AMX instructions can be executed and the XSAVE feature set can be used to manage TILEDATA.

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, or XCR0.Hi16_ZMM.
- Set either XCR0.BNDREG or 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.
- Set either XCR0.TILECFG or XCR0.TILEDATA while not setting the other.

After reset, all bits (except bit 0) in XCR0 are cleared to zero; XCR0[0] is set to 1.

2.7 PROTECTION-KEY RIGHTS REGISTERS (PKRU AND IA32_PKRS)

Processors may support either or both of two protection-key rights registers: PKRU for user-mode pages and the IA32_PKRS MSR (MSR index 6E1H) for supervisor-mode pages. 4-level paging and 5-level paging associate a 4-bit **protection key** with each page. The protection-key rights registers determine accessibility based on a page's protection key.

If CPUID.(EAX=07H,ECX=0H):ECX.PKU [bit 3] = 1, the processor supports the protection-key feature for user-mode pages. When CR4.PKE = 1, software can use the **protection-key rights register for user pages** (PKRU) to specify the access rights for user-mode pages for each protection key.

If CPUID.(EAX=07H,ECX=0H):ECX.PKS [bit 31] = 1, the processor supports the protection-key feature for supervisor-mode pages. When CR4.PKS = 1, software can use the **protection-key rights register for supervisor pages** (the IA32_PKRS MSR) to specify the access rights for supervisor-mode pages for each protection key.

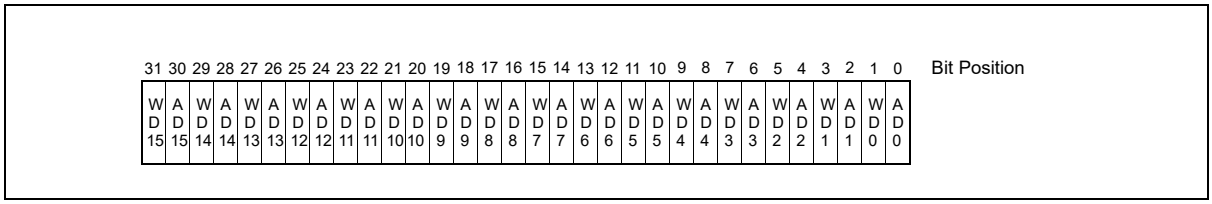


Figure 2-9. Format of Protection-Key Rights Registers

The format of each protection-key rights register is given in Figure 2-9. Each contains 16 pairs of disable controls to prevent data accesses to linear addresses (user-mode or supervisor-mode, depending on the register) based on their protection keys. Each protection key *i* ($0 \le i \le 15$) is associated with two bits in each protection-key rights register:

- Bit 2*i*, shown as "AD*i*" (access disable): if set, the processor prevents any data accesses to linear addresses (user-mode or supervisor-mode, depending on the register) with protection key *i.*
- Bit 2*i*+1, shown as "WD*i*" (write disable): if set, the processor prevents write accesses to linear addresses (user-mode or supervisor-mode, depending on the register) with protection key *i.*

(Bits 63:32 of the IA32_PKRS MSR are reserved and must be zero.)

See Section 5.6.2, "Protection Keys," for details of how the processor uses the protection-key rights registers to control accesses to linear addresses.

Software can read and write PKRU using the RDPKRU and WRPKRU instructions. The IA32_PKRS MSR can be read and written with the RDMSR and WRMSR instructions. Writes to the IA32_PKRS MSR using WRMSR are not serializing.

2.8 SYSTEM INSTRUCTION SUMMARY

System instructions handle system-level functions such as loading system registers, managing the cache, managing interrupts, or setting up the debug registers. Many of these instructions can be executed only by operating-system or executive procedures (that is, procedures running at privilege level 0). Others can be executed at any privilege level and are thus available to application programs.

Table 2-3 lists the system instructions and indicates whether they are available and useful for application programs. These instructions are described in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

Table 2-3. Summary of System Instructions

Instruction	Description	Useful to Application?	Protected from Application?
LLDT	Load LDT Register	No	Yes
SLDT	Store LDT Register	No	If CR4.UMIP = 1
LGDT	Load GDT Register	No	Yes
SGDT	Store GDT Register	No	If CR4.UMIP = 1
LTR	Load Task Register	No	Yes
STR	Store Task Register	No	If CR4.UMIP = 1
LIDT	Load IDT Register	No	Yes
SIDT	Store IDT Register	No	If CR4.UMIP = 1
MOV CR*n*	Load and store control registers	No	Yes
SMSW	Store MSW	Yes	If CR4.UMIP = 1
LMSW	Load MSW	No	Yes
CLTS	Clear TS flag in CR0	No	Yes
ARPL	Adjust RPL	Yes[1, 5]	No
LAR	Load Access Rights	Yes	No
LSL	Load Segment Limit	Yes	No
VERR	Verify for Reading	Yes	No
VERW	Verify for Writing	Yes	No
MOV DR*n*	Load and store debug registers	No	Yes
INVD	Invalidate cache, no writeback	No	Yes
WBINVD	Invalidate cache, with writeback	No	Yes
INVLPG	Invalidate TLB entry	No	Yes
HLT	Halt Processor	No	Yes
LOCK (Prefix)	Bus Lock	Yes	No
RSM	Return from system management mode	No	Yes
RDMSR[3]	Read Model-Specific Registers	No	Yes
WRMSR[3]	Write Model-Specific Registers	No	Yes
RDPMC[4]	Read Performance-Monitoring Counter	Yes	Yes[2]
RDTSC[3]	Read Time-Stamp Counter	Yes	Yes[2]
RDTSCP[7]	Read Serialized Time-Stamp Counter	Yes	Yes[2]
XGETBV	Return the state of XCR0	Yes	No
XSETBV	Enable one or more processor extended states	No[6]	Yes

Table 2-3. Summary of System Instructions (Contd.)

Instruction	Description	Useful to Application?	Protected from Application?

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 6.10.4, "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 6.10.1, "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 6.10.3, "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 6.10.2, "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.

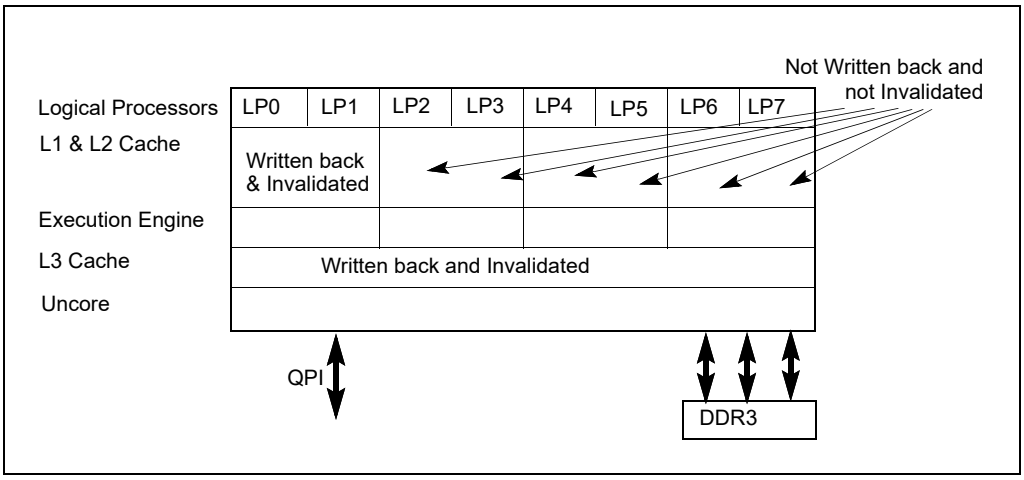


Figure 2-10. WBINVD Invalidation of Shared and Non-Shared Cache Hierarchy

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 20, "Last Branch Records."

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 at: https://perfmon-events.intel.com/, and the width/number of fixed-function counters are enumerated by CPUID leaf 0AH.

The time-stamp counter is a model-specific 64-bit counter that is reset to zero each time the processor is reset. If not reset, the counter will increment ~9.5×10^{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 21.1, "Performance Monitoring Overview," and Section 19.17, "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 11.4, "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).

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 6, "Protection," (for a description of the processor's protection mechanism) and Chapter 22, "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 ensures 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.

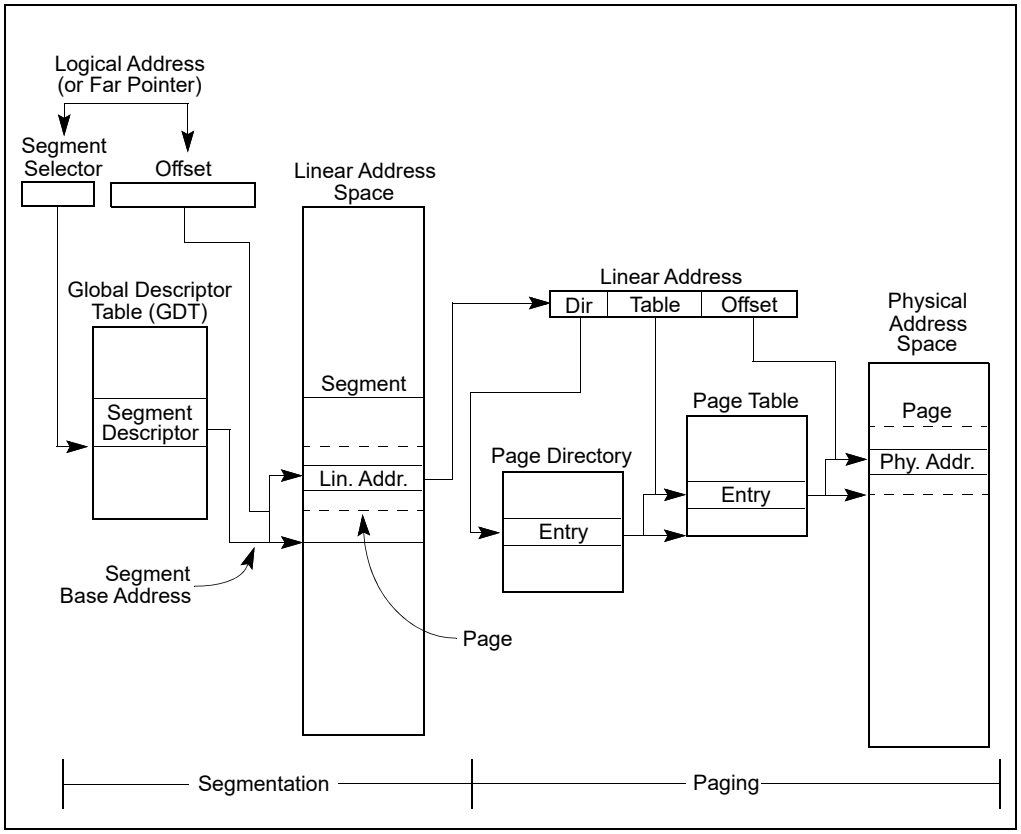


Figure 3-1. Segmentation and Paging

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.

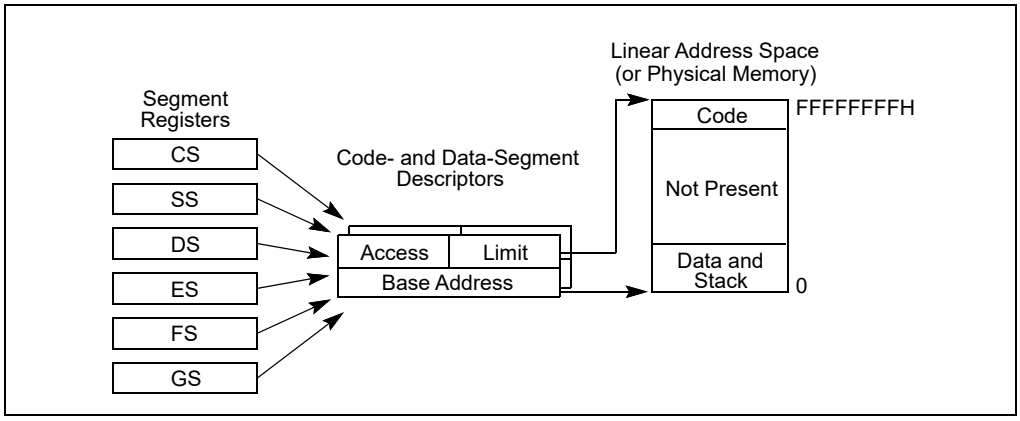


Figure 3-2. Flat Model

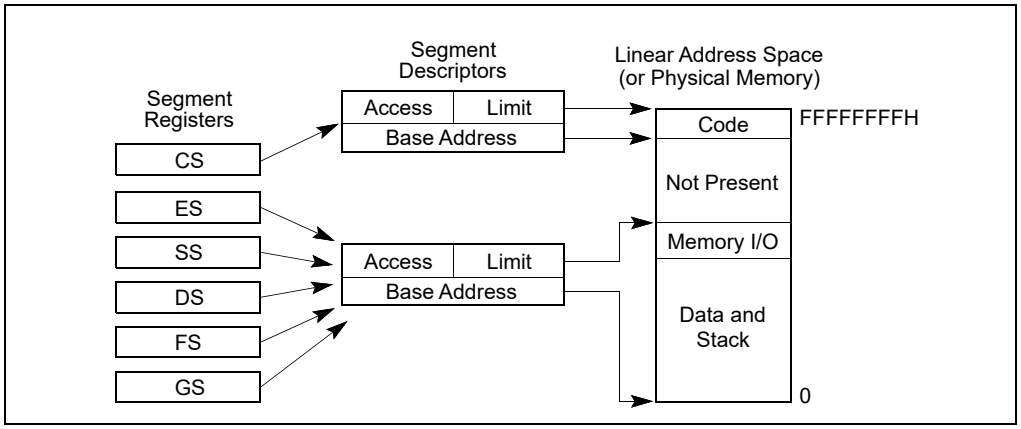


Figure 3-3. Protected Flat Model

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.

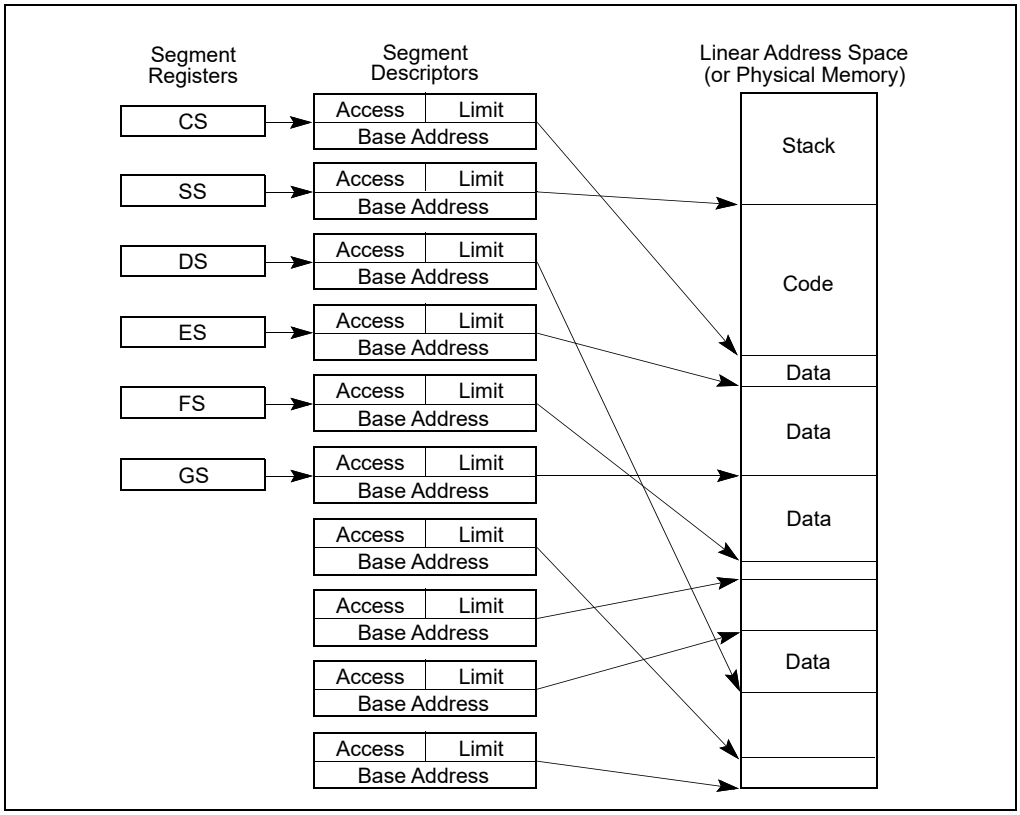


Figure 3-4. Multi-Segment Model

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 FFFFFFFFFH. 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 5, "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.80000001H:EDX[29] = 1), the size of the physical address range is implementation-specific and indicated by CPUID.80000008H: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 5, "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 ensure 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.

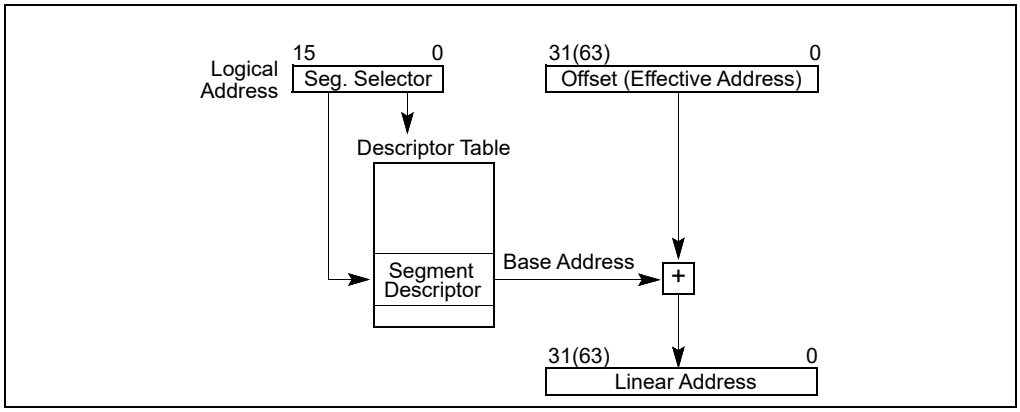


Figure 3-5. Logical Address to Linear Address Translation

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 5, "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 linear-address pre-processing (see Chapter 4).

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.

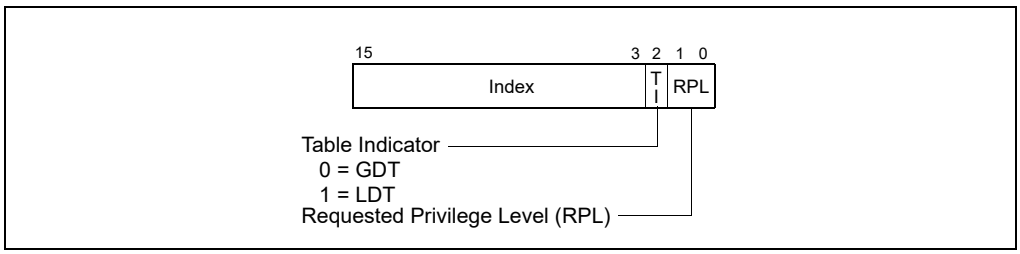


Figure 3-6. Segment Selector

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 6.5, "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.

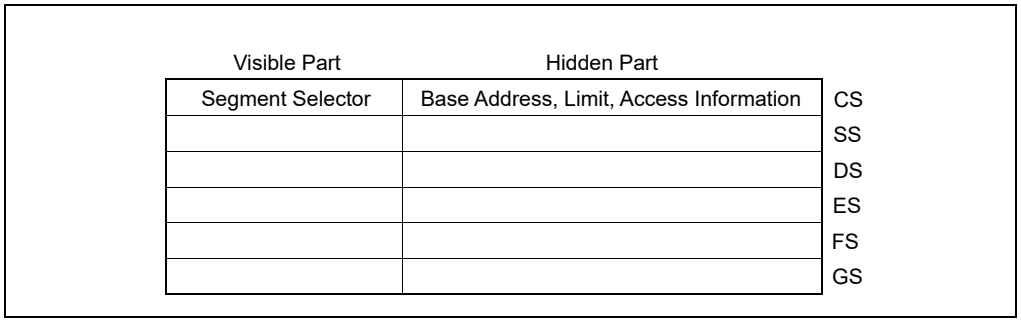


Figure 3-7. Segment Registers

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 performs linear-address pre-processing (Chapter 4) 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 address is subject to linear-address pre-processing.

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. See Section 4.5.2.

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; see Section 4.5.2.

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.

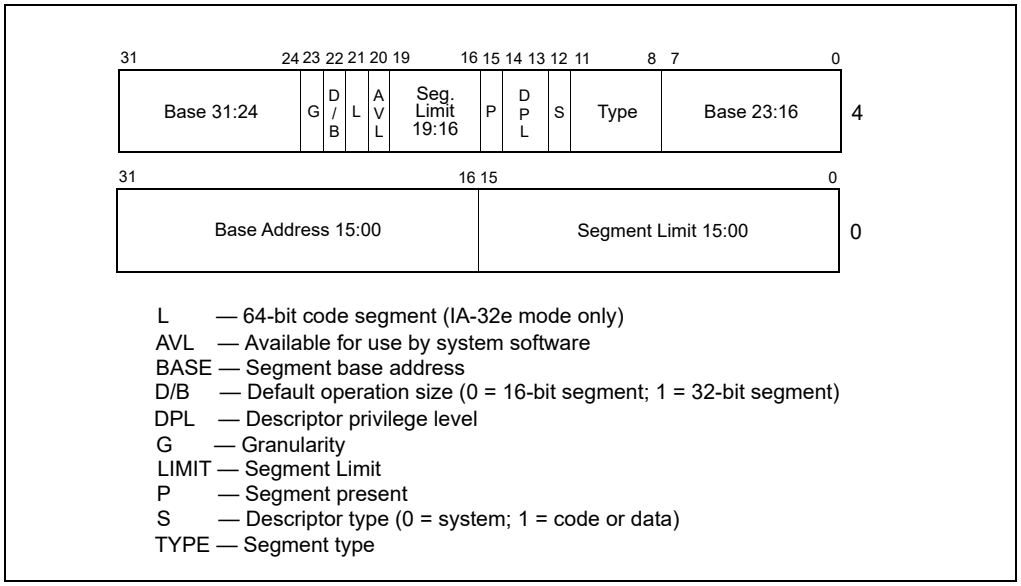


Figure 3-8. Segment Descriptor

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 6-1). 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 6.5, "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 FFFFFFFFH (4 GBytes); if the flag is clear, the upper bound is FFFFH (64 KBytes).

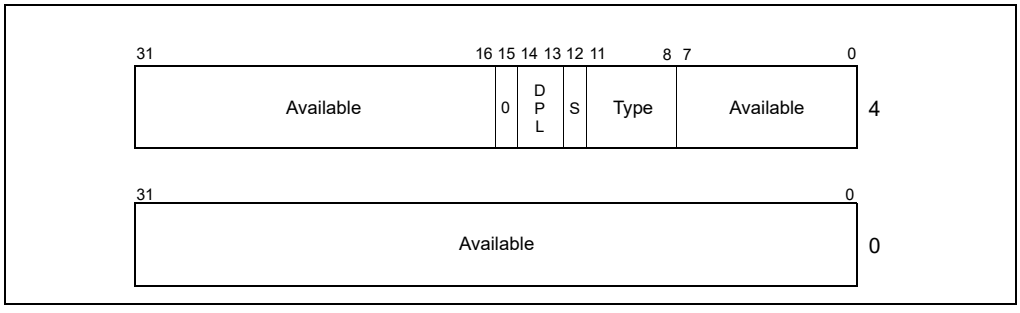


Figure 3-9. Segment Descriptor When Segment-Present Flag Is Clear

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 the L-bit is set, then the D-bit must be cleared. Bit 21 is not used outside IA-32e mode (or for data segments). Because an attempt to activate IA-32e mode will fault if the current CS has the L-bit set (see Section 11.8.5), software operating outside IA-32e mode should avoid loading CS from a descriptor that sets the L-bit.

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-enable (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-enable bit.

Table 3-1. Code- and Data-Segment Types

Type Field					Descriptor Type	Description
Decimal	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
		C	R	A		
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 6.8.1, "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 9.2.2, “TSS Descriptor,” (for more information on the system-segment descriptors); see Section 6.8.3, “Call Gates,” Section 7.11, “IDT Descriptors,” and Section 9.2.5, “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).

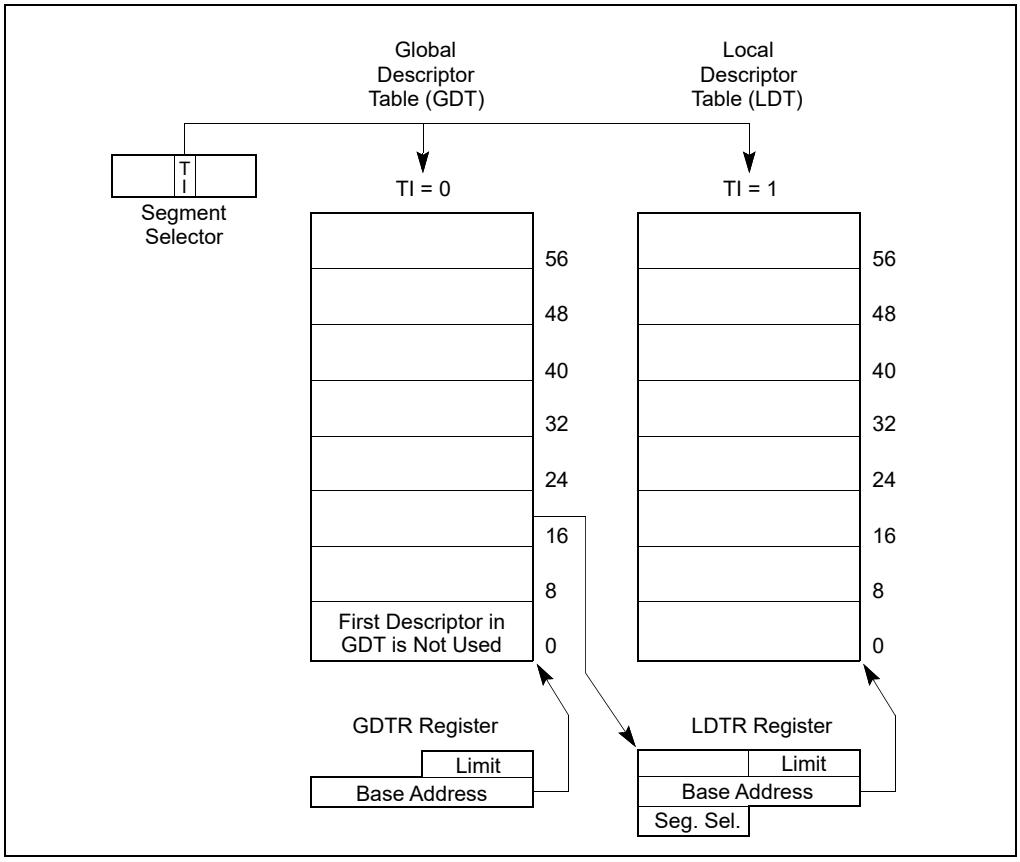


Figure 3-10. Global and Local Descriptor Tables

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).

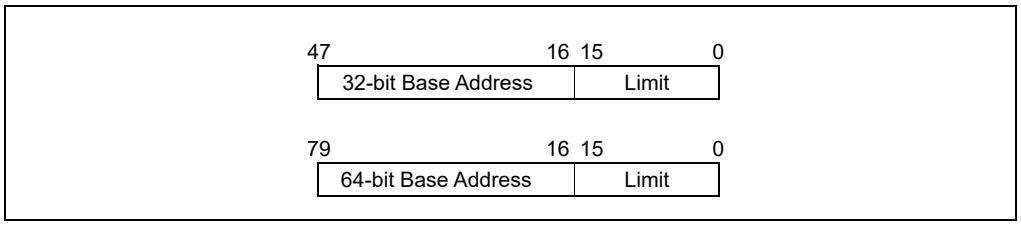


Figure 3-11. Pseudo-Descriptor Formats

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 6.8.3.1, “IA-32e Mode Call Gates”).
- IDT gate descriptors (see Section 7.14.1, “64-Bit Mode IDT”).
- LDT and TSS descriptors (see Section 9.2.3, “TSS Descriptor in 64-bit mode”).

CHAPTER 4
LINEAR-ADDRESS PRE-PROCESSING

As described in Chapter 3, "Protected-Mode Memory Management," software accesses to memory typically use logical addresses. The processor uses segmentation, as detailed in Section 3.4, to generate linear addresses from logical addresses. Linear addresses are then translated to physical addresses using paging, as described in Chapter 5, "Paging."

In IA-32e mode (if IA32_EFER.LMA = 1), linear addresses may undergo some pre-processing before being translated through paging.[1] Some of this pre-processing is done only if enabled by software, but some occurs unconditionally. Specifically, linear addresses are subject to pre-processing in IA-32e mode as follows:

1. **Linear-address-space separation (LASS).** This is a feature that, when enabled by software, may limit the linear addresses that are accessible by software, generating faults for accesses out of range.
2. **Linear-address masking (LAM).** This is a feature that, when enabled by software, masks certain linear-address bits.
3. **Canonicality checking.** As will be detailed in Chapter 5, paging does not translate all 64 bits of a linear address. Each linear address must be **canonical**, meaning that the untranslated bits have a fixed value. Memory accesses using a non-canonical address generate faults.

Both LASS and canonicality checking can generate faults. For any specific memory access, the two features generate the same fault. For that reason, the relative order of that checking is not defined and cannot be determined by software.

4.1 ENABLING AND ENUMERATION

Software enables LASS by setting CR4.LASS[bit 27]. Enabling of LAM is based on three different bits: CR3.LAM_U48[bit 62], CR3.LAM_U57[bit 61], and CR4.LAM_SUP[bit 28]. The use of these bits is explained in Section 4.4. Canonicality checking is not enabled by software and is always performed in 64-bit mode.

The processor enumerates support for LASS with CPUID.(EAX=07H,ECX=1):EAX.LASS[bit 6]. If this bit is enumerated as 1, software can set CR4.LASS.

The processor enumerates support for LAM with CPUID.(EAX=07H,ECX=1):EAX.LAM[bit 26]. If this bit is enumerated as 1, software can set CR3.LAM_U48, CR3.LAM_U57, and CR4.LAM_SUP.

4.2 MODE-BASED ACCESSES AND LINEAR-ADDRESS-SPACE PARTITIONING

Every access to a linear address is either a **supervisor-mode access** or a **user-mode access**. For all instruction fetches and most data accesses, this distinction is determined by the current privilege level (CPL): accesses made while CPL < 3 are supervisor-mode accesses, while accesses made while CPL = 3 are user-mode accesses.

Some operations implicitly access system data structures with linear addresses; the resulting accesses to those data structures are supervisor-mode accesses regardless of CPL. Such accesses include the following: accesses to the global descriptor table (GDT) or local descriptor table (LDT) to load a segment descriptor; accesses to the interrupt descriptor table (IDT) when delivering an interrupt or exception; accesses to the task-state segment (TSS) as part of a task switch or change of CPL; and accesses to a user posted-interrupt descriptor (UPID) during user-interrupt notification processing. Such accesses are called **implicit supervisor-mode accesses** regardless of CPL. Other accesses made while CPL < 3 are called **explicit supervisor-mode accesses**.[2]

1. The presentation in this chapter focuses on 64-bit addresses. 32-bit and 16-bit addresses can also be used in IA-32e mode. For the purposes of this chapter, the upper bits of such addresses (32 bits and 48 bits, respectively) are treated as if they were all zero.
2. The WRUSS instruction is an exception; although it can be executed only if CPL = 0, the processor treats its shadow-stack accesses as user-mode accesses.

Some 64-bit operating systems partition the 64-bit linear-address space into a supervisor portion and a user portion. Specifically, the upper half of the linear-address space (comprising addresses in which bit 63 is 1) is used for supervisor instructions and data, while the lower half (addresses in which bit 63 is 0) is for user instructions and data.

The LASS and LAM features are designed for operating systems that establish such linear-address-space partitioning. However, the features are defined and may be used even if such partitioning is not in effect.

4.3 LINEAR-ADDRESS-SPACE SEPARATION (LASS)

The access rights determined by paging (see Section 5.6) are based on whether a linear address is a supervisor-mode address or a user-mode address. Paging provides protection by preventing user-mode accesses to supervisor-mode addresses; in addition, there are paging features that can prevent supervisor-mode accesses to user-mode addresses.

These paging-based protections prevent malicious software from directly reading or writing memory inappropriately. However, they require the processor to traverse a hierarchy of paging structures in memory. Unprivileged software may be able to use the timing information resulting from this traversal to determine details about the paging structures, the layout of supervisor memory, or its use by supervisor software.

Linear-address-space separation (LASS) is an independent mechanism that can enforce mode-based protection without traversing the paging structures. Because LASS provides this protection as part of linear-address pre-processing, unprivileged software is denied paging-based timing information.

An operating system can use LASS to provide protections corresponding to the mode-based paging protections if it has established the linear-address-space partitioning outlined in Section 4.2.

4.3.1 Enumeration and Enabling

The processor enumerates support for LASS with CPUID.(EAX=07H,ECX=1):EAX.LASS[bit 6].

Software enables LASS by setting CR4.LASS[bit 27]. CR4.LASS can be set to 1 if CPUID.(EAX=07H,ECX=1):EAX.LASS[bit 6] is enumerated as 1.

The operation of LASS is also affected by the paging-mode bit CR4.SMAP[bit 21], which enables supervisor-access prevention. LASS enforces the equivalent of supervisor-mode execution prevention regardless of the setting of CR4.SMEP[bit 17]. See Section 4.3.2 for details.

4.3.2 Operation of Linear-Address-Space Separation (LASS)

This section describes the operation of linear-address-space separation (LASS). This operation applies only in IA-32e mode (if IA32_EFER.LMA = 1) and only if CR4.LASS = 1.

As indicated earlier, LASS enforces protections similar to those enforced by paging. Violations of these protections are called **LASS violations**.

LASS violations typically result in faults. In most cases, an access causing a LASS violation results in a general protection exception (#GP); for stack accesses (those due to stack-oriented instructions, as well as accesses that implicitly or explicitly use the SS segment register), a stack fault (#SS) is generated. In either case, a null error code is produced.

Some accesses are not subject to faults due to LASS violations. These include prefetches (e.g., those resulting from execution of one of the PREFETCH*h* instructions), executions of the CLDEMOTE instruction, and accesses resulting from the speculative fetch or execution of an instruction. Such an access may cause a LASS violation; if it does, the access is not performed but no fault occurs.

The remainder of this section describes how LASS applies to different types of accesses to linear addresses. The items below discuss specific LASS violations based on bit 63 of a linear address. For a linear address with only 32 bits (or 16 bits), the processor treats bit 63 as if it were 0; this includes accesses in compatibility mode.

- A user-mode data access causes a LASS violation if it would access a linear address of which bit 63 is 1.

- A supervisor-mode data access causes a LASS violation if it would access a linear address of which bit 63 is 0, supervisor-mode access protection is enabled (by setting CR4.SMAP), and either RFLAGS.AC = 0 or the access is an implicit supervisor-mode access.
- A user-mode instruction fetch causes a LASS violation if it would fetch an instruction using a linear address of which bit 63 is 1.
- A supervisor-mode instruction fetch causes a LASS violation if it would accesses a linear address of which bit 63 is 0. (Unlike paging, this behavior of LASS applies regardless of the setting of CR4.SMEP.)

LASS for instruction fetches applies when the linear address in RIP is used to load an instruction from memory. Unlike canonicality checking (see Section 4.5.2), LASS does **not** apply to branch instructions that load RIP. A branch instruction can load RIP with an address that would violate LASS. Only when the address is used to fetch an instruction will a LASS violation occur, generating a #GP. (The return instruction pointer of the #GP handler is the address that incurred the LASS violation.)

4.4 LINEAR-ADDRESS MASKING

This section describes **linear-address masking** (**LAM**). LAM modifies linear addresses before they are subject to canonicality checking as described in Section 4.5. Doing so allows untranslated linear-address bits to contain arbitrary values.

In IA-32e mode, linear address have 64 bits and are translated either with 4-level paging, which translates the low 48 bits of each linear address, or with 5-level paging, which translates 57 bits. The upper linear-address bits are reserved by canonicality checking (see Section 4.5).

Software usages that associate metadata with a pointer might benefit from being able to place metadata in the upper (untranslated) bits of the pointer itself. However, the canonicality enforcement mentioned earlier implies that software would have to mask the metadata bits in a pointer (making it canonical) before using it as a linear address to access memory. LAM allows software to use pointers with metadata without having to mask the metadata bits. With LAM enabled, the processor masks the metadata bits in a pointer before using it as a linear address to access memory.

LAM applies only in 64-bit mode and only to addresses used for data accesses. It does not apply to addresses used for instruction fetches or to those being loaded into the RIP register (e.g., as targets of jump and call instructions).

4.4.1 Enumeration, Enabling, and Configuration

The processor enumerates support for LAM with CPUID.(EAX=07H,ECX=1):EAX.LAM[bit 26].

Enabling and configuration of LAM is controlled by the following control-register bits: CR3.LAM_U48[bit 62], CR3.LAM_U57[bit 61], and CR4.LAM_SUP[bit 28]. The use of these control bits is explained below.

LAM supports configurations that differ regarding which linear-address bits are masked and can be used for metadata. With **LAM48**, linear-address bits 62:48 are masked (resulting in a **LAM width** of 15); with **LAM57**, linear-address bits 62:57 are masked (a LAM width of 6).

Like LASS, LAM was designed for operating systems that establish the linear-address-space partitioning outlined in Section 4.2: linear addresses that clear bit 63 are used for user memory, while those that set bit 63 are for supervisor memory. For LAM, the identification of an address as user or supervisor is based solely on the value of bit 63 and does not depend on the CPL.

LAM and the LAM width can be configured independently for user and supervisor addresses (as identified in the previous paragraph, using bit 63). CR3.LAM_U48 and CR3.LAM_U57 enable and configure LAM for user addresses:

- If CR3.LAM_U48 = CR3.LAM_U57 = 0, LAM is not enabled for user addresses.
- If CR3.LAM_U48 = 1 and CR3.LAM_U57 = 0, LAM48 is enabled for user addresses (a LAM width of 15).
- If CR3.LAM_U57 = 1, LAM57 is enabled for user addresses (a LAM width of 6; CR3.LAM_U48 is ignored in this case).

CR4.LAM_SUP enables and configures LAM for supervisor addresses:

- If CR4.LAM_SUP = 0, LAM is not enabled for supervisor addresses.

- If CR4.LAM_SUP = 1, LAM is enabled for supervisor addresses with a width determined by the paging mode (see Section 5.1.1):
  - If 4-level paging is enabled, LAM48 is enabled for supervisor addresses (a LAM width of 15).
  - If 5-level paging is enabled, LAM57 is enabled for supervisor addresses (a LAM width of 6).

4.4.2 Treatment of Data Accesses with LAM Active

When LAM is active, linear addresses used to access data are **masked** before they are subject to the canonicality checking identified in Section 4.5. Specifically, LAM modifies a linear address by extending the value of one address bit (depending on the LAM width) over others:

- When LAM48 is enabled (see Section 4.4.1), the processor modifies each linear address to replace each of bits 62:48 with the value of bit 47.
- When LAM57 is enabled, each of bits 62:57 is replaced by the value of bit 56 (bits 56:48 are not modified).

4.4.3 Paging Interactions

As explained in Section 4.4.2, LAM masks certain bits in a linear address before that address is translated by paging.

In most cases, the address bits in the masked positions are not used by address translation. However, if 5-level paging is active and LAM48 is enabled for user pointers, bit 47 of a user pointer is extended over bits 62:48 to form a linear address, and bits 56:48 are used by 5-level paging.

Page faults report the faulting linear address in CR2. Because LAM masking applies before paging, the faulting linear address recorded in CR2 reflects the result of that masking and does not contain any masked metadata.

The INVLPG, INVPCID, and INVVPID instructions can be used to invalidate any translation lookaside buffer (TLB) entries for specified linear addresses. LAM does not apply to those addresses, although those addresses are subject to canonicality checking (see Section 4.5.4).

4.5 CANONICALITY CHECKING

Memory accesses in IA-32e mode can use 64-bit linear addresses. As detailed in Section 5.5.4, 4-level paging translates the low 48 bits of each linear address, while 5-level paging translates the low 57 bits. The remaining upper bits (bits 63:48 with 4-level paging; bits 63:57 with 5-level paging) are not translated.

IA-32e mode accounts for the fact that address bits are not translated (and thus should be reserved) with the concept of **canonicality**. In general, a linear address is **canonical** if the untranslated bits are a sign-extension of the most significant translated bit. More specifically, there are two types of canonicality:

- A linear address is **paging canonical** if it is canonical for the current paging mode: a linear address is canonical for 4-level paging (**48-bit canonical**) if bits 63:47 of the address are identical; it is canonical for 5-level paging (**57-bit canonical**) if bits 63:56 are identical.
- A linear address is **CPU canonical** if it is canonical relative to the widest linear address supported by the processor: 48-bit canonical if the processor supports only 4-level paging and 57-bit canonical if the processor supports 5-level paging.

Unlike LASS and LAM, there is no control to enable canonicality checking. It always applies (as described in this section) when 64-bit linear addresses are used.

Section 4.5.1 and Section 4.5.2 explain canonicality checking for accesses to memory using linear addresses and for loads of the instruction pointer, respectively. Section 4.5.3 details how canonicality checking applies to certain system registers that contain linear addresses. Section 4.5.4 explains the role of canonicality checking by instructions that invalidate TLB entries.

NOTE

Section 4.5.2 and Section 4.5.3 discuss the canonicality checking performed by the WRMSR instruction. The WRMSRLIST and WRMSRNS instructions perform the same canonicality checking in corresponding situations. (Similarly, the characterization of RDMSR in Section 4.5.3 applies also to RDMSRLIST.)

4.5.1 Memory Accesses

An access to memory using a linear address is allowed only if the address is paging canonical; if it is not, a **canonicality violation** occurs. In most cases, an access causing a canonicality violation results in a general protection exception (#GP); for stack accesses (those due to stack-oriented instructions, as well as accesses that implicitly or explicitly use the SS segment register), a stack fault (#SS) is generated. In either case, a null error code is produced.

When LAM is enabled, canonicality checking is performed after masking of the linear address. This implies that the requirements of canonicality on an original (unmasked) linear address used to access data are effectively relaxed when LAM is enabled:

- With LAM48, bit 63 and bit 47 of the original linear address must be identical.
- With LAM57 and 4-level paging, bit 63 and bits 56:47 of the original linear address must be identical.
- With LAM57 and 5-level paging, bit 63 and bit 56 of the original linear address must be identical.

While LAM applies only to data accesses, canonicality checking applies both data accesses and instruction fetches.

4.5.2 Loads of the Instruction Pointer (RIP)

In 64-bit mode, the RIP register contains the linear address of the instruction pointer. Operations that load RIP (including both instructions such as JMP as well as control transfers through the IDT) check first whether the value to be loaded is paging canonical. If it is not, the operation does not modify RIP and instead causes a #GP. (This #GP is delivered as a fault, meaning that the return instruction pointer of the fault handler is the address of the faulting instruction and not the non-canonical address whose load was attempted.)

This treatment applies to the SYSRET and SYSEXIT instructions, which load RIP from RCX and RDX, respectively. The SYSCALL and SYSENTER instructions load RIP from the IA32_LSTAR and IA32_SYSENTER_EIP MSRs, respectively. On processors that support only 4-level paging, these instructions do not check explicitly that the values being loaded are paging canonical. This is because the WRMSR instruction ensures that these MSRs necessarily contain values that are CPU canonical, which is the same as being paging canonical on processors that support only 4-level paging. On processors that support 5-level paging, the checking by WRMSR is relaxed to ensure only 57-bit canonicality. (See Section 4.5.3 for the treatment of WRMSR.) On such processors, an execution of SYSCALL or SYSENTER with 4-level paging checks that the value being loaded into RIP is 48-bit canonical (paging canonical).

4.5.3 System Registers Containing Linear Addresses

In addition to RIP, the CPU maintains numerous other registers that hold linear addresses:

- GDTR and IDTR (in their base-address portions).
- LDTR, TR, FS, and GS (in the base-address portions of their hidden descriptor caches).
- Control register CR2, which holds the linear address causing a page fault.
- The debug-address registers (DR0 through DR3), which hold the linear addresses of breakpoints.
- The following MSRs: IA32_BNDCFGS, IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_INTERRUPT_SSP_TABLE_ADDR, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_PL0_SSP, IA32_PL1_SSP, IA32_PL2_SSP, IA32_PL3_SSP, 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, IA32_S_CET, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP, IA32_UINTR_HANDLER, IA32_UINTR_PD, IA32_UINTR_STACKADJUST, IA32_U_CET, and IA32_UINTR_TT.
- The x87 FPU instruction pointer (FIP).

- The user-mode configuration register BNDCFGU, used by Intel® MPX.

With a few exceptions, the processor ensures that the addresses in these registers are always canonical in the following ways:

- Some instructions fault on attempts to load a linear-address register with a non-canonical address:
  - — An execution of the LGDT or LIDT instruction causes a general-protection exception (#GP) if the base address specified in the instruction's memory operand is not canonical.
  - — An execution of the LLDT or LTR instruction causes a #GP if the base address to be loaded from the GDT is not canonical.
  - — An execution of WRFSBASE or WRGSBASE causes a #GP if it would load the base address of either FS or GS with a non-canonical address.
  - — An execution of WRMSR causes a #GP if it would load any of the following MSRs with a non-canonical address: IA32_BNDCFGS, IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_INTERRUPT_SSP_TABLE_ADDR, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_PL0_SSP, IA32_PL1_SSP, IA32_PL2_SSP, IA32_PL3_SSP, 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, IA32_S_CET, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP, IA32_UINTR_HANDLER, IA32_UINTR_PD, IA32_UINTR_STACKADJUST, IA32_U_CET, and IA32_UINTR_TT.[1]
  - — An execution of XRSTORS causes a #GP if it would load any of the following MSRs with a non-canonical address: IA32_PL0_SSP, IA32_PL1_SSP, IA32_PL2_SSP, IA32_PL3_SSP, 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, IA32_U_CET, IA32_UINTR_HANDLER, IA32_UINTR_PD, IA32_UINTR_STACKADJUST, or IA32_UINTR_TT.

  With a small number of exceptions, this enforcement checks for CPU canonicality and is thus independent of the current paging mode. Thus, a processor that supports 5-level paging will allow the instructions mentioned above to load these registers with addresses that are 57-bit canonical but not 48-bit canonical, even if 4-level paging is active. (As a result, instructions that store these values — SGDT, SIDT, SLDT, STR, RDFSBASE, RDGSBASE, RDMSR, XSAVE, XSAVEC, XSAVEOPT, and XSAVES — may save addresses that are 57-bit canonical but not 48-bit canonical, even if 4-level paging is active.)

  The WRFSBASE and WRGSBASE instructions, which load the base address of FS and GS, respectively, operate differently. An execution of either of these instructions causes a #GP if it would load a base address with an address that is not paging canonical. Thus, if 4-level paging is active, these instructions do not allow loading of addresses that are 57-bit canonical but not 48-bit canonical.
- The FXRSTOR, XRSTOR, and XRSTORS instructions ignore attempts to load some of these registers with non-canonical addresses:
  - — Loads of FIP ignore any bits in the memory image beyond the enumerated maximum linear-address width. The processor sign-extends the most significant bit (e.g., bit 56 on processors that support 5-level paging) to ensure that FIP is always CPU canonical.
  - — Loads of BNDCFGU (by XRSTOR or XRSTORS) ignore any bits in the memory image beyond the enumerated maximum linear-address width. The processor sign-extends the most significant bit to ensure that BNDCFGU is always CPU canonical.
- Every non-control x87 instruction loads FIP. The value loaded is always paging canonical.
- CR2 can be loaded with the MOV to CR instruction. The instruction allows that register to be loaded with a non-canonical address. The MOV from CR instruction will return for CR2 the value last loaded into that register by a page fault or with the MOV to CR instruction, even if (for the latter case) the address is not canonical. Page faults load CR2 only with linear addresses that are paging canonical.
- DR0 through DR3 can be loaded with the MOV to DR instruction. The instruction allows those registers to be loaded with non-canonical addresses. The MOV from DR instruction will return for a debug register the value

1. Such canonicality checking may apply also when the WRMSR instruction is used to load some non-architectural MSRs (not listed here) that hold a linear address.

last loaded into that register with the MOV to DR instruction, even if the address is not canonical. Breakpoint address matching is supported only for linear addresses that are paging canonical.

4.5.4 TLB-Invalidation Instructions

The Intel 64 architecture includes three instructions that may invalidate TLB entries for the linear address of an instruction operand: INVLPG, INVPCID, and INVVPID. The following items describe how they are affected by canonicality.

- The INVLPG instruction takes a memory operand. It invalidates any TLB entries that the logical processor is caching for the linear address of that operand for the current linear address space. The instruction does not fault if that address is not paging canonical. However, no invalidation is performed because the processor does not cache TLB entries for addresses that are not paging canonical.
- The INVPCID instruction takes a register operand (INVPCID type) and a memory operand (INVPCID descriptor). If the INVPCID type is 0, the instruction invalidates any TLB entries that the logical processor is caching for the linear address and PCID specified in the INVPCID descriptor. If the linear address is not CPU canonical, the instruction causes a #GP. If the processor supports 5-level paging, the instruction will not cause such a #GP for an address that is 57-bit canonical, regardless of paging mode, even if 4-level paging is active and the address is not 48-bit canonical.
- The INVVPID instruction takes a register operand (INVVPID type) and a memory operand (INVVPID descriptor). If the INVPCID type is 0, the instruction invalidates any TLB entries that the logical processor is caching for the linear address and VPID specified in the INVVPID descriptor. If the linear address is not CPU canonical, the instruction fails.[1] If the processor supports 5-level paging, the instruction will not fail for an address that is 57-bit canonical, regardless of paging mode, even if 4-level paging is active and the address is not 48-bit canonical.

LAM does not apply to the linear addresses that these instructions use to invalidate TLB entries.

1. INVVPID is a VMX instruction. In response to certain conditions, execution of a VMX instruction may **fail**, meaning that it does not complete its normal operation. When a VMX instruction fails, control passes to the next instruction (rather than to a fault handler) and a flag is set to report the failure.

CHAPTER 5
PAGING

Chapter 3 explains how segmentation converts logical addresses to linear addresses. **Paging** (or linear-address translation) is the process of translating linear addresses so that they can be used to access memory or I/O devices. Paging translates each linear address to a **physical address** and determines, for each translation, what accesses to the linear address are allowed (the address's **access rights**) and the type of caching used for such accesses (the address's **memory type**).

Intel-64 processors support four different paging modes. These modes are identified and defined in Section 5.1. Section 5.2 gives an overview of the translation mechanism that is used in all modes. Section 5.3, Section 5.4, and Section 5.5 discuss the four paging modes in detail.

Section 5.6 details how paging determines and uses access rights. Section 5.7 discusses exceptions that may be generated by paging (page-fault exceptions). Section 5.8 considers data which the processor writes in response to linear-address accesses (accessed and dirty flags).

Section 5.9 describes how paging determines the memory types used for accesses to linear addresses. Section 5.10 provides details of how a processor may cache information about linear-address translation. Section 5.11 outlines interactions between paging and certain VMX features. Section 5.12 gives an overview of how paging can be used to implement virtual memory.

5.1 PAGING MODES AND CONTROL BITS

Paging behavior is controlled by the following control bits:

- The WP and PG flags in control register CR0 (bit 16 and bit 31, respectively).
- The PSE, PAE, PGE, LA57, PCIDE, SMEP, SMAP, PKE, CET, and PKS flags in control register CR4 (bit 4, bit 5, bit 7, bit 12, bit 17, bit 20, bit 21, bit 22, bit 23, and bit 24, respectively).
- The LME and NXE flags in the IA32_EFER MSR (bit 8 and bit 11, respectively).
- The AC flag in the EFLAGS register (bit 18).
- The "enable HLAT" VM-execution control (tertiary processor-based VM-execution control bit 1; see Section 26.6.2, "Processor-Based VM-Execution Controls," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C).

Software enables paging by using the MOV to CR0 instruction to set CR0.PG. Before doing so, software should ensure control register CR3 contains the physical address of the first paging structure that the processor will use for linear-address translation (see Section 5.2) and that structure is initialized as desired. See Table 5-3, Table 5-7, and Table 5-12 for the use of CR3 in the different paging modes.

Section 5.1.1 describes how the values of CR0.PG, CR4.PAE, CR4.LA57, and IA32_EFER.LME determine whether paging is enabled and, if so, which of four paging modes is in use. Section 5.1.2 explains how to manage these bits to establish or make changes in paging modes. Section 5.1.3 discusses how CR0.WP, CR4.PSE, CR4.PGE, CR4.PCIDE, CR4.SMEP, CR4.SMAP, CR4.PKE, CR4.CET, CR4.PKS, and IA32_EFER.NXE modify the operation of the different paging modes.

5.1.1 Four Paging Modes

If CR0.PG = 0, paging is not used. The logical processor treats all linear addresses as if they were physical addresses. CR4.PAE, CR4.LA57, and IA32_EFER.LME are ignored by the processor, as are CR0.WP, CR4.PSE, CR4.PGE, CR4.SMEP, CR4.SMAP, and IA32_EFER.NXE. (CR4.CET is also ignored insofar as it affects linear-address access rights.)

Paging is enabled if CR0.PG = 1. Paging can be enabled only if protection is enabled (CR0.PE = 1). If paging is enabled, one of four paging modes is used. The values of CR4.PAE, CR4.LA57, and IA32_EFER.LME determine which paging mode is used:

- If CR4.PAE = 0, **32-bit paging** is used. 32-bit paging is detailed in Section 5.3. 32-bit paging uses CR0.WP, CR4.PSE, CR4.PGE, CR4.SMEP, CR4.SMAP, and CR4.CET as described in Section 5.1.3 and Section 5.6.
- If CR4.PAE = 1 and IA32_EFER.LME = 0, **PAE paging** is used. PAE paging is detailed in Section 5.4. PAE paging uses CR0.WP, CR4.PGE, CR4.SMEP, CR4.SMAP, CR4.CET, and IA32_EFER.NXE as described in Section 5.1.3 and Section 5.6.
- If CR4.PAE = 1, IA32_EFER.LME = 1, and CR4.LA57 = 0, **4-level paging**[1] is used.[2] 4-level paging is detailed in Section 5.5 (along with 5-level paging). 4-level paging uses CR0.WP, CR4.PGE, CR4.PCIDE, CR4.SMEP, CR4.SMAP, CR4.PKE, CR4.CET, CR4.PKS, and IA32_EFER.NXE as described in Section 5.1.3 and Section 5.6.
- If CR4.PAE = 1, IA32_EFER.LME = 1, and CR4.LA57 = 1, **5-level paging** is used. 5-level paging is detailed in Section 5.5 (along with 4-level paging). 5-level paging uses CR0.WP, CR4.PGE, CR4.PCIDE, CR4.SMEP, CR4.SMAP, CR4.PKE, CR4.CET, CR4.PKS, and IA32_EFER.NXE as described in Section 5.1.3 and Section 5.6.

NOTE

32-bit paging and PAE paging can be used only in legacy protected mode (IA32_EFER.LME = 0). In contrast, 4-level paging and 5-level paging can be used only IA-32e mode (IA32_EFER.LME = 1).

The four paging modes differ with regard to the following details:

- Linear-address width. The size of the linear addresses that can be translated.
- Physical-address width. The size of the physical addresses produced by paging.
- Page size. The granularity at which linear addresses are translated. Linear addresses on the same page are translated to corresponding physical addresses on the same page.
- Support for execute-disable access rights. In some paging modes, software can be prevented from fetching instructions from pages that are otherwise readable.
- Support for PCIDs. With 4-level paging and 5-level paging, software can enable a facility by which a logical processor caches information for multiple linear-address spaces. The processor may retain cached information when software switches between different linear-address spaces.
- Support for protection keys. With 4-level paging and 5-level paging, each linear address is associated with a **protection key**. Software can use the protection-key rights registers to disable, for each protection key, how certain accesses to linear addresses associated with that protection key.

Table 5-1 illustrates the principal differences between the four paging modes.

Table 5-1. Properties of Different Paging Modes

Paging Mode	PG in CR0	PAE in CR4	LME in IA32_EFER	LA57 in CR4	Lin.-Addr. Width	Phys.-Addr. Width[1]	Page Sizes	Supports Execute-Disable?	Supports PCIDs and protection keys?
None	0	N/A	N/A	N/A	32	32	N/A	No	No
32-bit	1	0	0[2]	N/A	32	Up to 40[3]	4 KB 4 MB[4]	No	No
PAE	1	1	0	N/A	32	Up to 52	4 KB 2 MB	Yes[5]	No
4-level	1	1	1	0	48	Up to 52	4 KB 2 MB 1 GB[6]	Yes[5]	Yes[7]

1. Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.
2. The LMA flag in the IA32_EFER MSR (bit 10) is a status bit that indicates whether the logical processor is in IA-32e mode (and thus uses either 4-level paging or 5-level paging). The processor always sets IA32_EFER.LMA to CR0.PG & IA32_EFER.LME. Software cannot directly modify IA32_EFER.LMA; an execution of WRMSR to the IA32_EFER MSR ignores bit 10 of its source operand.

Table 5-1. Properties of Different Paging Modes (Contd.)

Paging Mode	PG in CR0	PAE in CR4	LME in IA32_EFER	LA57 in CR4	Lin.-Addr. Width	Phys.-Addr. Width[1]	Page Sizes	Supports Execute-Disable?	Supports PCIDs and protection keys?
5-level	1	1	1	1	57	Up to 52	4 KB 2 MB 1 GB[6]	Yes[5]	Yes[7]

NOTES:

1. The physical-address width is always bounded by MAXPHYADDR; see Section 5.1.4.
2. The processor ensures that IA32_EFER.LME must be 0 if CR0.PG = 1 and CR4.PAE = 0.
3. 32-bit paging supports physical-address widths of more than 32 bits only for 4-MByte pages and only if the PSE-36 mechanism is supported; see Section 5.1.4 and Section 5.3.
4. 32-bit paging uses 4-MByte pages only if CR4.PSE = 1; see Section 5.3.
5. Execute-disable access rights are applied only if IA32_EFER.NXE = 1; see Section 5.6.
6. Processors that support 4-level paging or 5-level paging do not necessarily support 1-GByte pages; see Section 5.1.4.
7. PCIDs are used only if CR4.PCIDE = 1; see Section 5.10.1. Protection keys are used only if certain conditions hold; see Section 5.6.2.

Because 32-bit paging and PAE paging are used only in legacy protected mode and because legacy protected mode cannot produce linear addresses larger than 32 bits, 32-bit paging and PAE paging translate 32-bit linear addresses.

4-level paging and 5-level paging are used only in IA-32e mode. IA-32e mode has two sub-modes:

- Compatibility mode. This sub-mode uses only 32-bit linear addresses. In this sub-mode, 4-level paging and 5-level paging treat bits 63:32 of such an address as all 0. These addresses are subject to linear-address pre-processing, specifically linear-address-space separation (Section 4.3).
- 64-bit mode. This sub-mode produces 64-bit linear addresses. These addresses are then subject to linear-address pre-processing (Chapter 4). As part of this, the processor enforces **canonicality** (Section 4.5), ensuring that the upper bits of such an address are identical: bits 63:47 for 4-level paging and bits 63:56 for 5-level paging. 4-level paging (respectively, 5-level paging) does not use bits 63:48 (respectively, bits 63:57) of such addresses.

5.1.2 Paging-Mode Enabling

If CR0.PG = 1, a logical processor is in one of four paging modes, depending on the values of CR4.PAE, IA32_EFER.LME, and CR4.LA57. Figure 5-1 illustrates how software can enable these modes and make transitions between them. The following items identify certain limitations and other details:

- IA32_EFER.LME cannot be modified while paging is enabled (CR0.PG = 1). Attempts to do so using WRMSR cause a general-protection exception (#GP(0)).
- Paging cannot be enabled (by setting CR0.PG to 1) while CR4.PAE = 0 and IA32_EFER.LME = 1. Attempts to do so using MOV to CR0 cause a general-protection exception (#GP(0)).
- One node in Figure 5-1 is labeled "IA-32e mode." This node represents either 4-level paging (if CR4.LA57 = 0) or 5-level paging (if CR4.LA57 = 1). As noted in the following items, software cannot modify CR4.LA57 (effecting transition between 4-level paging and 5-level paging) without first disabling paging.
- CR4.PAE and CR4.LA57 cannot be modified while either 4-level paging or 5-level paging is in use (when CR0.PG = 1 and IA32_EFER.LME = 1). Attempts to do so using MOV to CR4 cause a general-protection exception (#GP(0)).
- Regardless of the current paging mode, software can disable paging by clearing CR0.PG with MOV to CR0.[1]

1. If the logical processor is in 64-bit mode or if CR4.PCIDE = 1, an attempt to clear CR0.PG causes a general-protection exception (#GP). Software should transition to compatibility mode and clear CR4.PCIDE before attempting to disable paging.

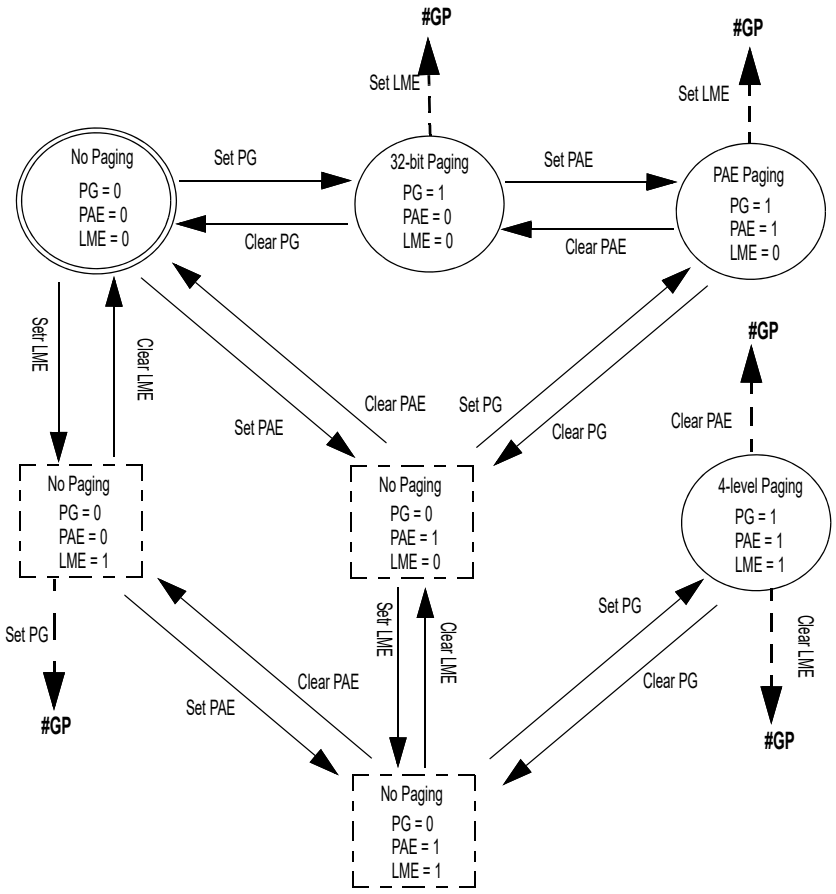


Figure 5-1. Enabling and Changing Paging Modes

- Software can transition between 32-bit paging and PAE paging by changing the value of CR4.PAE with MOV to CR4.
- Software cannot transition directly between 4-level paging (or 5-level paging) and any of other paging mode. It must first disable paging (by clearing CR0.PG with MOV to CR0), then set CR4.PAE, IA32_EFER.LME, and CR4.LA57 to the desired values (with MOV to CR4 and WRMSR), and then re-enable paging (by setting CR0.PG with MOV to CR0). As noted earlier, an attempt to modify CR4.PAE, IA32_EFER.LME, or CR.LA57 while 4-level paging or 5-level paging is enabled causes a general-protection exception (#GP(0)).
- VMX transitions allow transitions between paging modes that are not possible using MOV to CR or WRMSR. This is because VMX transitions can load CR0, CR4, and IA32_EFER in one operation. See Section 5.11.1.

5.1.3 Paging-Mode Modifiers

Details of how each paging mode operates are determined by the following control bits:

- The WP flag in CR0 (bit 16).
- The PSE, PGE, PCIDE, SMEP, SMAP, PKE, CET, and PKS flags in CR4 (bit 4, bit 7, bit 17, bit 20, bit 21, bit 22, bit 23, and bit 24, respectively).
- The NXE flag in the IA32_EFER MSR (bit 11).
- The "enable HLAT" VM-execution control (tertiary processor-based VM-execution control bit 1).

CR0.WP allows pages to be protected from supervisor-mode writes. If CR0.WP = 0, supervisor-mode write accesses are allowed to linear addresses with read-only access rights; if CR0.WP = 1, they are not. (User-mode write accesses are never allowed to linear addresses with read-only access rights, regardless of the value of CR0.WP.) Section 5.6 explains how access rights are determined, including the definition of supervisor-mode and user-mode accesses.

CR4.PSE enables 4-MByte pages for 32-bit paging. If CR4.PSE = 0, 32-bit paging can use only 4-KByte pages; if CR4.PSE = 1, 32-bit paging can use both 4-KByte pages and 4-MByte pages. See Section 5.3 for more information. (PAE paging, 4-level paging, and 5-level paging can use multiple page sizes regardless of the value of CR4.PSE.)

CR4.PGE enables global pages. If CR4.PGE = 0, no translations are shared across address spaces; if CR4.PGE = 1, specified translations may be shared across address spaces. See Section 5.10.2.4 for more information.

CR4.PCIDE enables process-context identifiers (PCIDs) for 4-level paging and 5-level paging. PCIDs allow a logical processor to cache information for multiple linear-address spaces. See Section 5.10.1 for more information.

CR4.SMEP allows pages to be protected from supervisor-mode instruction fetches. If CR4.SMEP = 1, software operating in supervisor mode cannot fetch instructions from linear addresses that are accessible in user mode. Section 5.6 explains how access rights are determined, including the definition of supervisor-mode accesses and user-mode accessibility.

CR4.SMAP allows pages to be protected from supervisor-mode data accesses. If CR4.SMAP = 1, software operating in supervisor mode cannot access data at linear addresses that are accessible in user mode. Software can override this protection by setting EFLAGS.AC. Section 5.6 explains how access rights are determined, including the definition of supervisor-mode accesses and user-mode accessibility.

CR4.PKE and CR4.PKS enable specification of access rights based on **protection keys**. 4-level paging and 5-level paging associate each linear address with a protection key. When CR4.PKE = 1, the PKRU register specifies, for each protection key, whether user-mode linear addresses with that protection key can be read or written. When CR4.PKS = 1, the IA32_PKRS MSR does the same for supervisor-mode linear addresses. See Section 5.6 for more information.

CR4.CET enables **control-flow enforcement technology**, including the shadow-stack feature. If CR4.CET = 1, certain memory accesses are identified as **shadow-stack accesses** and certain linear addresses translate to **shadow-stack pages**. Section 5.6 explains how access rights are determined for these accesses and pages. (The processor allows CR4.CET to be set only if CR0.WP is also set.)

IA32_EFER.NXE enables execute-disable access rights for PAE paging, 4-level paging, and 5-level paging. If IA32_EFER.NXE = 1, instruction fetches can be prevented from specified linear addresses (even if data reads from the addresses are allowed). Section 5.6 explains how access rights are determined. (IA32_EFER.NXE has no effect with 32-bit paging. Software that wants to use this feature to limit instruction fetches from readable pages must use PAE paging, 4-level paging, or 5-level paging.)

The "enable HLAT" VM-execution control enables **HLAT paging** for 4-level paging and 5-level paging. HLAT paging does not use control register CR3 to identify the address of the first paging structure used for linear-address translation; instead, that structure is located using a field in the virtual-machine control structure (VMCS). In addition, HLAT paging interprets certain bits in paging-structure entries differently than ordinary paging. See Section 5.5 for details.

5.1.4 Enumeration of Paging Features by CPUID

Software can discover support for different paging features using the CPUID instruction:

- PSE: page-size extensions for 32-bit paging.
  If CPUID.01H:EDX.PSE [bit 3] = 1, CR4.PSE may be set to 1, enabling support for 4-MByte pages with 32-bit paging (see Section 5.3).
- PAE: physical-address extension.
  If CPUID.01H:EDX.PAE [bit 6] = 1, CR4.PAE may be set to 1, enabling PAE paging (this setting is also required for 4-level paging and 5-level paging).
- PGE: global-page support.
  If CPUID.01H:EDX.PGE [bit 13] = 1, CR4.PGE may be set to 1, enabling the global-page feature (see Section 5.10.2.4).

- PAT: page-attribute table.
  If CPUID.01H:EDX.PAT [bit 16] = 1, the 8-entry page-attribute table (PAT) is supported. When the PAT is supported, three bits in certain paging-structure entries select a memory type (used to determine type of caching used) from the PAT (see Section 5.9.2).
- PSE-36: page-size extensions with 40-bit physical-address extension.
  If CPUID.01H:EDX.PSE-36 [bit 17] = 1, the PSE-36 mechanism is supported, indicating that translations using 4-MByte pages with 32-bit paging may produce physical addresses with up to 40 bits (see Section 5.3).
- PCID: process-context identifiers.
  If CPUID.01H:ECX.PCID [bit 17] = 1, CR4.PCIDE may be set to 1, enabling process-context identifiers (see Section 5.10.1).
- SMEP: supervisor-mode execution prevention.
  If CPUID.(EAX=07H,ECX=0H):EBX.SMEP [bit 7] = 1, CR4.SMEP may be set to 1, enabling supervisor-mode execution prevention (see Section 5.6).
- SMAP: supervisor-mode access prevention.
  If CPUID.(EAX=07H,ECX=0H):EBX.SMAP [bit 20] = 1, CR4.SMAP may be set to 1, enabling supervisor-mode access prevention (see Section 5.6).
- PKU: protection keys for user-mode pages.
  If CPUID.(EAX=07H,ECX=0H):ECX.PKU [bit 3] = 1, CR4.PKE may be set to 1, enabling protection keys for user-mode pages (see Section 5.6).
- OSPKE: enabling of protection keys for user-mode pages.
  CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4] returns the value of CR4.PKE. Thus, protection keys for user-mode pages are enabled if this flag is 1 (see Section 5.6).
- CET: control-flow enforcement technology.
  If CPUID.(EAX=07H,ECX=0H):ECX.CET_SS [bit 7] = 1, CR4.CET may be set to 1, enabling shadow-stack pages (see Section 5.6).
- LA57: 57-bit linear addresses and 5-level paging.
  If CPUID.(EAX=07H,ECX=0):ECX.LA57 [bit 16] = 1, CR4.LA57 may be set to 1, enabling 5-level paging.
- PKS: protection keys for supervisor-mode pages.
  If CPUID.(EAX=07H,ECX=0H):ECX.PKS [bit 31] = 1, CR4.PKS may be set to 1, enabling protection keys for supervisor-mode pages (see Section 5.6).
- NX: execute disable.
  If CPUID.80000001H:EDX.NX [bit 20] = 1, IA32_EFER.NXE may be set to 1, allowing software to disable execute access to selected pages (see Section 5.6). (Processors that do not support CPUID function 80000001H do not allow IA32_EFER.NXE to be set to 1.)
- Page1GB: 1-GByte pages.
  If CPUID.80000001H:EDX.Page1GB [bit 26] = 1, 1-GByte pages may be supported with 4-level paging and 5-level paging (see Section 5.5).
- LM: IA-32e mode support.
  If CPUID.80000001H:EDX.LM [bit 29] = 1, IA32_EFER.LME may be set to 1, enabling IA-32e mode (with either 4-level paging or 5-level paging). (Processors that do not support CPUID function 80000001H do not allow IA32_EFER.LME to be set to 1.)
- CPUID.80000008H:EAX[7:0] reports the physical-address width supported by the processor. (For processors that do not support CPUID function 80000008H, the width is generally 36 if CPUID.01H:EDX.PAE [bit 6] = 1 and 32 otherwise.) This width is referred to as MAXPHYADDR. MAXPHYADDR is at most 52.
- CPUID.80000008H:EAX[15:8] reports the linear-address width supported by the processor. Generally, this value is reported as follows:
  - If CPUID.80000001H:EDX.LM [bit 29] = 0, the value is reported as 32.
  - If CPUID.80000001H:EDX.LM [bit 29] = 1 and CPUID.(EAX=07H,ECX=0):ECX.LA57 [bit 16] = 0, the value is reported as 48.
  - If CPUID.(EAX=07H,ECX=0):ECX.LA57 [bit 16] = 1, the value is reported as 57.

  (Processors that do not support CPUID function 80000008H, support a linear-address width of 32.)

5.2 HIERARCHICAL PAGING STRUCTURES: AN OVERVIEW

All four paging modes translate linear addresses using **hierarchical paging structures**. This section provides an overview of their operation. Section 5.3, Section 5.4, Section 5.5, and Section 5.6 provide details for the four paging modes.

Every paging structure is 4096 Bytes in size and comprises a number of individual **entries**. With 32-bit paging, each entry is 32 bits (4 bytes); there are thus 1024 entries in each structure. With the other paging modes, each entry is 64 bits (8 bytes); there are thus 512 entries in each structure. (PAE paging includes one exception, a paging structure that is 32 bytes in size, containing 4 64-bit entries.)

The processor uses the upper portion of a linear address to identify a series of paging-structure entries. The last of these entries identifies the physical address of the region to which the linear address translates (called the **page frame**). The lower portion of the linear address (called the **page offset**) identifies the specific address within that region to which the linear address translates.

Each paging-structure entry contains a physical address, which is either the address of another paging structure or the address of a page frame. In the first case, the entry is said to **reference** the other paging structure; in the latter, the entry is said to **map a page**.

The first paging structure used for any translation is located at the physical address in CR3.[1] A linear address is translated using the following iterative procedure. A portion of the linear address (initially the uppermost bits) selects an entry in a paging structure (initially the one located using CR3). If that entry references another paging structure, the process continues with that paging structure and with the portion of the linear address immediately below that just used. If instead the entry maps a page, the process completes: the physical address in the entry is that of the page frame and the remaining lower portion of the linear address is the page offset.

The following items give an example for each of the four paging modes (each example locates a 4-KByte page frame):

- With 32-bit paging, each paging structure comprises $1024 = 2^{10}$ entries. For this reason, the translation process uses 10 bits at a time from a 32-bit linear address. Bits 31:22 identify the first paging-structure entry and bits 21:12 identify a second. The latter identifies the page frame. Bits 11:0 of the linear address are the page offset within the 4-KByte page frame. (See Figure 5-2 for an illustration.)
- With PAE paging, the first paging structure comprises only $4 = 2^2$ entries. Translation thus begins by using bits 31:30 from a 32-bit linear address to identify the first paging-structure entry. Other paging structures comprise $512 = 2^9$ entries, so the process continues by using 9 bits at a time. Bits 29:21 identify a second paging-structure entry and bits 20:12 identify a third. This last identifies the page frame. (See Figure 5-5 for an illustration.)
- With 4-level paging, each paging structure comprises $512 = 2^9$ entries and translation uses 9 bits at a time from a 48-bit linear address. Bits 47:39 identify the first paging-structure entry, bits 38:30 identify a second, bits 29:21 a third, and bits 20:12 identify a fourth. Again, the last identifies the page frame. (See Figure 5-8 for an illustration.)
- 5-level paging is similar to 4-level paging except that 5-level paging translates 57-bit linear addresses. Bits 56:48 identify the first paging-structure entry, while the remaining bits are used as with 4-level paging.

The translation process in each of the examples above completes by identifying a page frame; the page frame is part of the **translation** of the original linear address. In some cases, however, the paging structures may be configured so that the translation process terminates before identifying a page frame. This occurs if the process encounters a paging-structure entry that is marked "not present" (because its P flag — bit 0 — is clear) or in which a reserved bit is set. In this case, there is no translation for the linear address; an access to that address causes a page-fault exception (see Section 5.7).

In the examples above, a paging-structure entry maps a page with a 4-KByte page frame when only 12 bits remain in the linear address; entries identified earlier always reference other paging structures. That may not apply in other cases. The following items identify when an entry maps a page and when it references another paging structure:

- If more than 12 bits remain in the linear address, bit 7 (PS — page size) of the current paging-structure entry is consulted. If the bit is 0, the entry references another paging structure; if the bit is 1, the entry maps a page.

1. If HLAT paging is in use, a different mechanism is used to identify the first paging structure. See Section 5.5 for more information.

- If only 12 bits remain in the linear address, the current paging-structure entry always maps a page (bit 7 is used for other purposes).

If a paging-structure entry maps a page when more than 12 bits remain in the linear address, the entry identifies a page frame larger than 4 KBytes. For example, 32-bit paging uses the upper 10 bits of a linear address to locate the first paging-structure entry; 22 bits remain. If that entry maps a page, the page frame is 2^{22} Bytes = 4 MBytes. 32-bit paging can use 4-MByte pages if CR4.PSE = 1. The other paging modes can use 2-MByte pages (regardless of the value of CR4.PSE). 4-level paging and 5-level paging can use 1-GByte pages if the processor supports them (see Section 5.1.4).

Paging structures are given different names based on their uses in the translation process. Table 5-2 gives the names of the different paging structures. It also provides, for each structure, the source of the physical address used to locate it (CR3 or a different paging-structure entry); the bits in the linear address used to select an entry from the structure; and details of whether and how such an entry can map a page.

Table 5-2. Paging Structures in the Different Paging Modes

Paging Structure	Entry Name	Paging Mode	Physical Address of Structure	Bits Selecting Entry	Page Mapping
PML5 table	PML5E	32-bit, PAE, 4-level	N/A		
		5-level	CR3[1]	56:48	N/A (PS must be 0)
PML4 table	PML4E	32-bit, PAE	N/A		
		4-level	CR3[1]	47:39	N/A (PS must be 0)
		5-level	PML5E		
Page-directory-pointer table	PDPTE	32-bit	N/A		
		PAE	CR3	31:30	N/A (PS must be 0)
		4-level, 5-level	PML4E	38:30	1-GByte page if PS=1[2]
Page directory	PDE	32-bit	CR3	31:22	4-MByte page if PS=1[3]
		PAE, 4-level, 5-level	PDPTE	29:21	2-MByte page if PS=1
Page table	PTE	32-bit	PDE	21:12	4-KByte page
		PAE, 4-level, 5-level		20:12	

NOTES:

1. If HLAT paging is in use, a different mechanism is used to identify the first paging structure. See Section 5.5 for more information.
2. Not all processors support 1-GByte pages; see Section 5.1.4.
3. 32-bit paging ignores the PS flag in a PDE (and uses the entry to reference a page table) unless CR4.PSE = 1. Not all processors support 4-MByte pages with 32-bit paging; see Section 5.1.4.

5.3 32-BIT PAGING

A logical processor uses 32-bit paging if CR0.PG = 1 and CR4.PAE = 0. 32-bit paging translates 32-bit linear addresses to 40-bit physical addresses.[1] Although 40 bits corresponds to 1 TByte, linear addresses are limited to 32 bits; at most 4 GBytes of linear-address space may be accessed at any given time.

32-bit paging uses a hierarchy of paging structures to produce a translation for a linear address. CR3 is used to locate the first paging-structure, the page directory. Table 5-3 illustrates how CR3 is used with 32-bit paging.

32-bit paging may map linear addresses to either 4-KByte pages or 4-MByte pages. Figure 5-2 illustrates the translation process when it uses a 4-KByte page; Figure 5-3 covers the case of a 4-MByte page. The following items describe the 32-bit paging process in more detail as well has how the page size is determined:

- A 4-KByte naturally aligned page directory is located at the physical address specified in bits 31:12 of CR3 (see Table 5-3). A page directory comprises 1024 32-bit entries (PDEs). A PDE is selected using the physical address defined as follows:
  - Bits 39:32 are all 0.
  - Bits 31:12 are from CR3.
  - Bits 11:2 are bits 31:22 of the linear address.
  - Bits 1:0 are 0.

Because a PDE is identified using bits 31:22 of the linear address, it controls access to a 4-Mbyte region of the linear-address space. Use of the PDE depends on CR4.PSE and the PDE's PS flag (bit 7):

- If CR4.PSE = 1 and the PDE's PS flag is 1, the PDE maps a 4-MByte page (see Table 5-4). The final physical address is computed as follows:
  - Bits 39:32 are bits 20:13 of the PDE.
  - Bits 31:22 are bits 31:22 of the PDE.[2]
  - Bits 21:0 are from the original linear address.
- If CR4.PSE = 0 or the PDE's PS flag is 0, a 4-KByte naturally aligned page table is located at the physical address specified in bits 31:12 of the PDE (see Table 5-5). A page table comprises 1024 32-bit entries (PTEs). A PTE is selected using the physical address defined as follows:
  - Bits 39:32 are all 0.
  - Bits 31:12 are from the PDE.
  - Bits 11:2 are bits 21:12 of the linear address.
  - Bits 1:0 are 0.
- Because a PTE is identified using bits 31:12 of the linear address, every PTE maps a 4-KByte page (see Table 5-6). The final physical address is computed as follows:
  - Bits 39:32 are all 0.
  - Bits 31:12 are from the PTE.
  - Bits 11:0 are from the original linear address.

If a paging-structure entry's P flag (bit 0) is 0 or if the entry sets any reserved bit, the entry is used neither to reference another paging-structure entry nor to map a page. There is no translation for a linear address whose translation would use such a paging-structure entry; a reference to such a linear address causes a page-fault exception (see Section 5.7).

---

1. Bits in the range 39:32 are 0 in any physical address used by 32-bit paging except those used to map 4-MByte pages. If the processor does not support the PSE-36 mechanism, this is true also for physical addresses used to map 4-MByte pages. If the processor does support the PSE-36 mechanism and MAXPHYADDR < 40, bits in the range 39:MAXPHYADDR are 0 in any physical address used to map a 4-MByte page. (The corresponding bits are reserved in PDEs.) See Section 5.1.4 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.
2. The upper bits in the final physical address do not all come from corresponding positions in the PDE; the physical-address bits in the PDE are not all contiguous.

With 32-bit paging, there are reserved bits only if CR4.PSE = 1:

- If the P flag and the PS flag (bit 7) of a PDE are both 1, the bits reserved depend on MAXPHYADDR, and whether the PSE-36 mechanism is supported:[1]
  - If the PSE-36 mechanism is not supported, bits 21:13 are reserved.
  - If the PSE-36 mechanism is supported, bits 21:(M–19) are reserved, where M is the minimum of 40 and MAXPHYADDR.
- If the PAT is not supported:[2]
  - If the P flag of a PTE is 1, bit 7 is reserved.
  - If the P flag and the PS flag of a PDE are both 1, bit 12 is reserved.

(If CR4.PSE = 0, no bits are reserved with 32-bit paging.)

A reference using a linear address that is successfully translated to a physical address is performed only if allowed by the access rights of the translation; see Section 5.6.

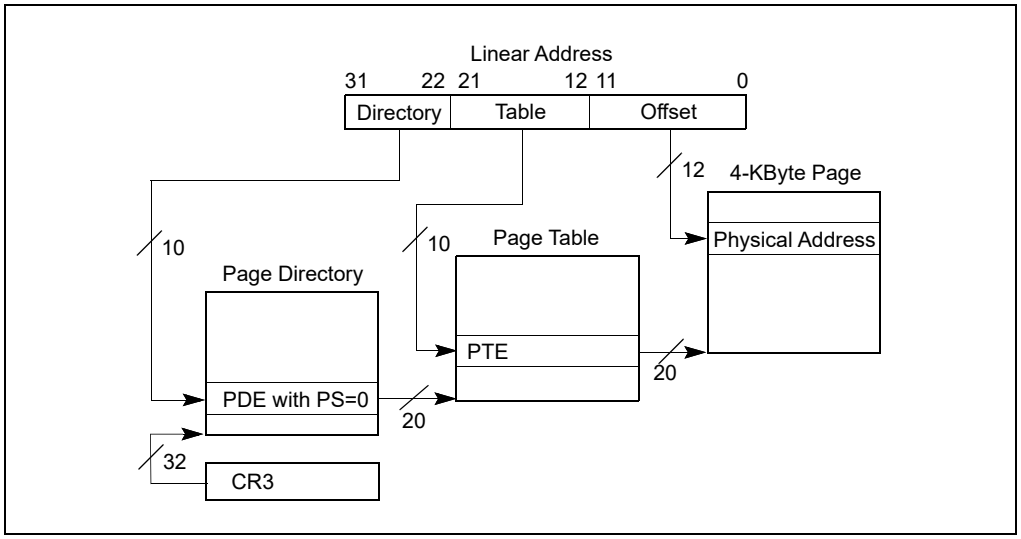


Figure 5-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

1. See Section 5.1.4 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.
2. See Section 5.1.4 for how to determine whether the PAT is supported.

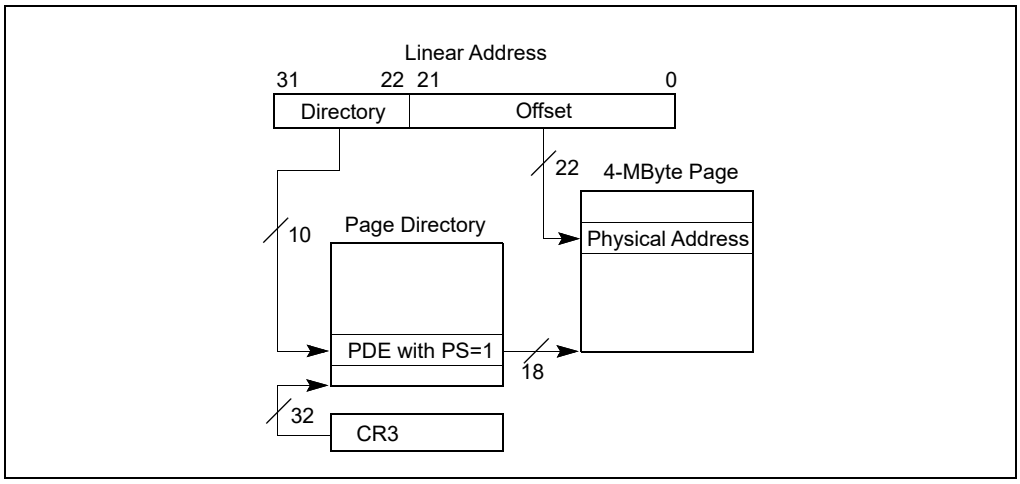


Figure 5-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

Figure 5-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are "not present"; bit 0 (P) and bit 7 (PS) are highlighted because they determine how such an entry is used.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			**CR3**
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)						Bits 39:32 of address[2]			PAT	Ignored			G	**1**	D	A	PCD	PWT	U/S	R/W	**1**	**PDE: 4MB page**
Address of page table																				Ignored				**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	**PDE: page table**
Ignored																															**0**	**PDE: not present**
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	**1**	**PTE: 4KB page**
Ignored																															**0**	**PTE: not present**

Figure 5-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

NOTES:

1. CR3 has 64 bits on processors supporting the Intel-64 architecture. These bits are ignored with 32-bit paging.
2. This example illustrates a processor in which MAXPHYADDR is 36. If this value is larger or smaller, the number of bits reserved in positions 20:13 of a PDE mapping a 4-MByte page will change.

Table 5-3. Use of CR3 with 32-Bit Paging

Bit Position(s)	Contents
2:0	Ignored
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 5.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 5.9)
11:5	Ignored
31:12	Physical address of the 4-KByte aligned page directory used for linear-address translation
63:32	Ignored (these bits exist only on processors supporting the Intel-64 architecture)

Table 5-4. Format of a 32-Bit Page-Directory Entry that Maps a 4-MByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-MByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte page referenced by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte page referenced by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 5.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 5.9)
5 (A)	Accessed; indicates whether software has accessed the 4-MByte page referenced by this entry (see Section 5.8)
6 (D)	Dirty; indicates whether software has written to the 4-MByte page referenced by this entry (see Section 5.8)
7 (PS)	Page size; must be 1 (otherwise, this entry references a page table; see Table 5-5)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 5.10); ignored otherwise
11:9	Ignored
12 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 5.9.2); otherwise, reserved (must be 0)[1]
(M–20):13	Bits (M–1):32 of physical address of the 4-MByte page referenced by this entry[2]
21:(M–19)	Reserved (must be 0)
31:22	Bits 31:22 of physical address of the 4-MByte page referenced by this entry

NOTES:

1. See Section 5.1.4 for how to determine whether the PAT is supported.
2. If the PSE-36 mechanism is not supported, M is 32, and this row does not apply. If the PSE-36 mechanism is supported, M is the minimum of 40 and MAXPHYADDR (this row does not apply if MAXPHYADDR = 32). See Section 5.1.4 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.

Table 5-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte region controlled by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 5.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 5.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 5.8)
6	Ignored
7 (PS)	If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page; see Table 5-4); otherwise, ignored
11:8	Ignored
31:12	Physical address of 4-KByte aligned page table referenced by this entry

Table 5-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 5.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 5.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 5.10); ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

NOTES:

1. See Section 5.1.4 for how to determine whether the PAT is supported.

5.4 PAE PAGING

A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 0. PAE paging translates 32-bit linear addresses to 52-bit physical addresses.[1] Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 32 bits; at most 4 GBytes of linear-address space may be accessed at any given time.

With PAE paging, a logical processor maintains a set of four (4) PDPTE registers, which are loaded from an address in CR3. Linear address are translated using 4 hierarchies of in-memory paging structures, each located using one of the PDPTE registers. (This is different from the other paging modes, in which there is one hierarchy referenced by CR3.)

Section 5.4.1 discusses the PDPTE registers. Section 5.4.2 describes linear-address translation with PAE paging.

5.4.1 PDPTE Registers

When PAE paging is used, CR3 references the base of a 32-Byte **page-directory-pointer table**. Table 5-7 illustrates how CR3 is used with PAE paging.

Table 5-7. Use of CR3 with PAE Paging

Bit Position(s)	Contents
4:0	Ignored
31:5	Physical address of the 32-Byte aligned page-directory-pointer table used for linear-address translation
63:32	Ignored (these bits exist only on processors supporting the Intel-64 architecture)

The page-directory-pointer-table comprises four (4) 64-bit entries called PDPTEs. Each PDPTE controls access to a 1-GByte region of the linear-address space. Corresponding to the PDPTEs, the logical processor maintains a set of four (4) internal, non-architectural PDPTE registers, called PDPTE0, PDPTE1, PDPTE2, and PDPTE3. The logical processor loads these registers from the PDPTEs in memory as part of certain operations:

- If PAE paging would be in use following an execution of MOV to CR0 or MOV to CR4 (see Section 5.1.1) and the instruction is modifying any of CR0.CD, CR0.NW, CR0.PG, CR4.PAE, CR4.PGE, CR4.PSE, or CR4.SMEP; then the PDPTEs are loaded from the address in CR3.
- If MOV to CR3 is executed while the logical processor is using PAE paging, the PDPTEs are loaded from the address being loaded into CR3.
- If PAE paging is in use and a task switch changes the value of CR3, the PDPTEs are loaded from the address in the new CR3 value.
- Certain VMX transitions load the PDPTE registers. See Section 5.11.1.

Table 5-8 gives the format of a PDPTE. If any of the PDPTEs sets both the P flag (bit 0) and any reserved bit, the MOV to CR instruction causes a general-protection exception (#GP(0)) and the PDPTEs are not loaded.[2] As shown in Table 5-8, bits 2:1, 8:5, and 63:MAXPHYADDR are reserved in the PDPTEs.

1. If MAXPHYADDR < 52, bits in the range 51:MAXPHYADDR will be 0 in any physical address used by PAE paging. (The corresponding bits are reserved in the paging-structure entries.) See Section 5.1.4 for how to determine MAXPHYADDR.
2. On some processors, reserved bits are checked even in PDPTEs in which the P flag (bit 0) is 0.

Table 5-8. Format of a PAE Page-Directory-Pointer-Table Entry (PDPTE)

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page directory
2:1	Reserved (must be 0)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 5.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 5.9)
8:5	Reserved (must be 0)
11:9	Ignored
(M–1):12	Physical address of 4-KByte aligned page directory referenced by this entry[1]
63:M	Reserved (must be 0)

NOTES:

1. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 5.1.4.

5.4.2 Linear-Address Translation with PAE Paging

PAE paging may map linear addresses to either 4-KByte pages or 2-MByte pages. Figure 5-5 illustrates the translation process when it produces a 4-KByte page; Figure 5-6 covers the case of a 2-MByte page. The following items describe the PAE paging process in more detail as well has how the page size is determined:

- Bits 31:30 of the linear address select a PDPTE register (see Section 5.4.1); this is PDPTE*i*, where *i* is the value of bits 31:30.[1] Because a PDPTE register is identified using bits 31:30 of the linear address, it controls access to a 1-GByte region of the linear-address space. If the P flag (bit 0) of PDPTE*i* is 0, the processor ignores bits 63:1, and there is no mapping for the 1-GByte region controlled by PDPTE*i*. A reference using a linear address in this region causes a page-fault exception (see Section 5.7).
- If the P flag of PDPTE*i* is 1, 4-KByte naturally aligned page directory is located at the physical address specified in bits 51:12 of PDPTE*i* (see Table 5-8 in Section 5.4.1). A page directory comprises 512 64-bit entries (PDEs). A PDE is selected using the physical address defined as follows:
  - Bits 51:12 are from PDPTE*i*.
  - Bits 11:3 are bits 29:21 of the linear address.
  - Bits 2:0 are 0.

Because a PDE is identified using bits 31:21 of the linear address, it controls access to a 2-Mbyte region of the linear-address space. Use of the PDE depends on its PS flag (bit 7):

- If the PDE's PS flag is 1, the PDE maps a 2-MByte page (see Table 5-9). The final physical address is computed as follows:
  - Bits 51:21 are from the PDE.
  - Bits 20:0 are from the original linear address.
- If the PDE's PS flag is 0, a 4-KByte naturally aligned page table is located at the physical address specified in bits 51:12 of the PDE (see Table 5-10). A page table comprises 512 64-bit entries (PTEs). A PTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PDE.

1. With PAE paging, the processor does not use CR3 when translating a linear address (as it does in the other paging modes). It does not access the PDPTEs in the page-directory-pointer table during linear-address translation.

  - Bits 11:3 are bits 20:12 of the linear address.
  - Bits 2:0 are 0.
- Because a PTE is identified using bits 31:12 of the linear address, every PTE maps a 4-KByte page (see Table 5-11). The final physical address is computed as follows:
  - Bits 51:12 are from the PTE.
  - Bits 11:0 are from the original linear address.

If the P flag (bit 0) of a PDE or a PTE is 0 or if a PDE or a PTE sets any reserved bit, the entry is used neither to reference another paging-structure entry nor to map a page. There is no translation for a linear address whose translation would use such a paging-structure entry; a reference to such a linear address causes a page-fault exception (see Section 5.7).

The following bits are reserved with PAE paging:

- If the P flag (bit 0) of a PDE or a PTE is 1, bits 62:MAXPHYADDR are reserved.
- If the P flag and the PS flag (bit 7) of a PDE are both 1, bits 20:13 are reserved.
- If IA32_EFER.NXE = 0 and the P flag of a PDE or a PTE is 1, the XD flag (bit 63) is reserved.
- If the PAT is not supported:[1]
  - If the P flag of a PTE is 1, bit 7 is reserved.
  - If the P flag and the PS flag of a PDE are both 1, bit 12 is reserved.

A reference using a linear address that is successfully translated to a physical address is performed only if allowed by the access rights of the translation; see Section 5.6.

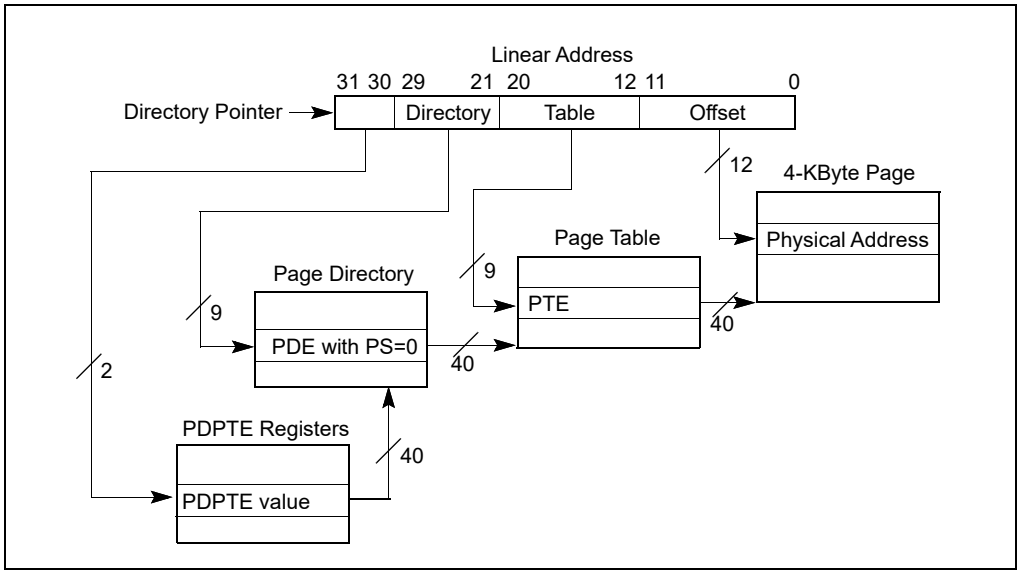


Figure 5-5. Linear-Address Translation to a 4-KByte Page using PAE Paging

1. See Section 5.1.4 for how to determine whether the PAT is supported.

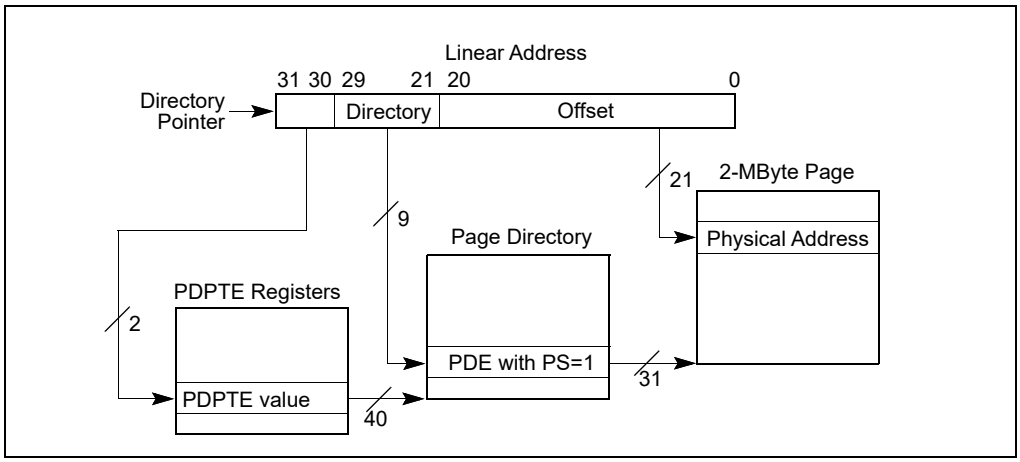


Figure 5-6. Linear-Address Translation to a 2-MByte Page using PAE Paging

Table 5-9. Format of a PAE Page-Directory Entry that Maps a 2-MByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 2-MByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 2-MByte page referenced by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte page referenced by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 5.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 5.9)
5 (A)	Accessed; indicates whether software has accessed the 2-MByte page referenced by this entry (see Section 5.8)
6 (D)	Dirty; indicates whether software has written to the 2-MByte page referenced by this entry (see Section 5.8)
7 (PS)	Page size; must be 1 (otherwise, this entry references a page table; see Table 5-10)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 5.10); ignored otherwise
11:9	Ignored
12 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 5.9.2); otherwise, reserved (must be 0)[1]
20:13	Reserved (must be 0)
(M–1):21	Physical address of the 2-MByte page referenced by this entry
62:M	Reserved (must be 0)
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte page controlled by this entry; see Section 5.6); otherwise, reserved (must be 0)

NOTES:

1. See Section 5.1.4 for how to determine whether the PAT is supported.

Table 5-10. Format of a PAE Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 2-MByte region controlled by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte region controlled by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 5.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 5.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 5.8)
6	Ignored
7 (PS)	Page size; must be 0 (otherwise, this entry maps a 2-MByte page; see Table 5-9)
11:8	Ignored
(M–1):12	Physical address of 4-KByte aligned page table referenced by this entry
62:M	Reserved (must be 0)
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte region controlled by this entry; see Section 5.6); otherwise, reserved (must be 0)

Table 5-11. Format of a PAE Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 5.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 5.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 5.10); ignored otherwise

Table 5-11. Format of a PAE Page-Table Entry that Maps a 4-KByte Page (Contd.)

Bit Position(s)	Contents
11:9	Ignored
(M–1):12	Physical address of the 4-KByte page referenced by this entry
62:M	Reserved (must be 0)
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 5.6); otherwise, reserved (must be 0)

NOTES:

1. See Section 5.1.4 for how to determine whether the PAT is supported.

Figure 5-7 gives a summary of the formats of CR3 and the paging-structure entries with PAE paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are "not present"; bit 0 (P) and bit 7 (PS) are highlighted because they determine how a paging-structure entry is used.

63	62–M	(M–1)–32	31–5	4	3	2	1	0	
Ignored[2]			Address of page-directory-pointer table						CR3
Reserved[3]			Address of page directory		Ign.	Rsvd.	PCD	PWT	Rsvd	**1**	PDPTE: present
Ignored										**0**	PDTPE: not present
XD[4]	Reserved		Address of 2MB page frame	Reserved	PAT	Ign.	G	**1**	D	A	PCD	PWT	U/S	R/W	**1**	PDE: 2MB page
XD	Reserved		Address of page table			Ign.		**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	PDE: page table
Ignored															**0**	PDE: not present
XD	Reserved		Address of 4KB page frame			Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	**1**	PTE: 4KB page
Ignored															**0**	PTE: not present

Figure 5-7. Formats of CR3 and Paging-Structure Entries with PAE Paging

NOTES:

1. M is an abbreviation for MAXPHYADDR.
2. CR3 has 64 bits only on processors supporting the Intel-64 architecture. These bits are ignored with PAE paging.
3. Reserved fields must be 0.
4. If IA32_EFER.NXE = 0 and the P flag of a PDE or a PTE is 1, the XD flag (bit 63) is reserved.

5.5 4-LEVEL PAGING AND 5-LEVEL PAGING

Because the operation of 4-level paging and 5-level paging is very similar, they are described together in this section. The following items highlight the distinctions between the two paging modes:

- A logical processor uses 4-level paging if CR0.PG = 1, CR4.PAE = 1, IA32_EFER.LME = 1, and CR4.LA57 = 0. 4-level paging translates 48-bit linear addresses to 52-bit physical addresses.[1] Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 48 bits; at most 256 TBytes of linear-address space may be accessed at any given time.
- A logical processor uses 5-level paging if CR0.PG = 1, CR4.PAE = 1, IA32_EFER.LME = 1, and CR4.LA57 = 1. 5-level paging translates 57-bit linear addresses to 52-bit physical addresses. Thus, 5-level paging supports a linear-address space sufficient to access the entire physical-address space.

5.5.1 Ordinary Paging and HLAT Paging

There are two forms of 4-level paging and 5-level paging that differ principally with regard to how linear-address translation identifies the first paging structure.

The normal form is called **ordinary paging**, and it uses CR3 to locate the first paging structure, similar to what is done for 32-bit paging. Section 5.5.2 provides details of this use of CR3.

An alternative form of paging may be used with the VMX feature called hypervisor-managed linear-address translation (HLAT). Called **HLAT paging**, this form is used only in VMX non-root operation and only if the "enable HLAT" VM-execution control is 1.[2] HLAT paging locates the first paging structure using a VM-execution control field in the VMCS called the **HLAT pointer** (**HLATP**). Section 5.5.3 provides details.

Whether HLAT paging is used to translate a specific linear address depends on the address and on the value of a VM-execution control field in the VMCS called the **HLAT prefix size**:

- If the HLAT prefix size is zero, every linear address is translated using HLAT paging.
- If the HLAT prefix size is not zero, a linear address is translated using HLAT paging if bit 63 of the address is 1.[3] The address is translated using ordinary paging if bit 63 of the address is 0.

In some cases, HLAT paging may specify that a translation of a linear address must be restarted. When this occurs, the linear address is then translated using ordinary paging (starting with a paging structure identified using CR3). The situations leading to this restart are detailed in Section 5.5.4, and additional details of the restart process are given in Section 5.5.5.

5.5.2 Use of CR3 with Ordinary 4-Level Paging and 5-Level Paging

Ordinary 4-level paging and 5-level paging each translate linear addresses using a hierarchy of in-memory paging structures located using the contents of CR3, which is used to locate the first paging structure. For 4-level paging, this is the PML4 table, and for 5-level paging it is the PML5 table. Use of CR3 with 4-level paging and 5-level paging depends on whether process-context identifiers (PCIDs) have been enabled by setting CR4.PCIDE:

- Table 5-12 illustrates how CR3 is used with 4-level paging and 5-level paging if CR4.PCIDE = 0.

Table 5-12. Use of CR3 with 4-Level Paging and 5-level Paging and CR4.PCIDE = 0

Bit Position(s)	Contents
2:0	Ignored

1. If MAXPHYADDR < 52, bits in the range 51:MAXPHYADDR will be 0 in any physical address used by 4-level paging. (The corresponding bits are reserved in the paging-structure entries.) See Section 5.1.4 for how to determine MAXPHYADDR.
2. HLAT paging is used only with 4-level paging and 5-level paging. It is never used with 32-bit paging or PAE paging, regardless of the value of the "enable HLAT" VM-execution control.
3. This behavior applies if the CPU enumerates a maximum HLAT prefix size of 1 in IA32_VMX_EPT_VPID_CAP[53:48] (see Appendix A.10). Behavior when a different value is enumerated is not currently defined.

Table 5-12. Use of CR3 with 4-Level Paging and 5-level Paging and CR4.PCIDE = 0 (Contd.)

Bit Position(s)	Contents
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the PML4 table or PML5 table during linear-address translation (see Section 5.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the PML4 table or PML5 table during linear-address translation (see Section 5.9.2)
11:5	Ignored
M–1:12	Physical address of the 4-KByte aligned PML4 table or PML5 table used for linear-address translation[1]
60:M	Reserved (must be 0)
61	Enables LAM57 for user pointers; Section 4.4.[2]
62	Enables LAM48 for user pointers; ignored if bit 61 is set.[2]
63	Reserved (must be 0)[3]

NOTES:

1. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 5.1.4.
2. LAM is not a paging feature.
3. See Section 5.10.4.1 for use of bit 63 of the source operand of the MOV to CR3 instruction.

- Table 5-13 illustrates how CR3 is used with 4-level paging and 5-level paging if CR4.PCIDE = 1.

Table 5-13. Use of CR3 with 4-Level Paging and 5-Level Paging and CR4.PCIDE = 1

Bit Position(s)	Contents
11:0	PCID (see Section 5.10.1)[1]
M–1:12	Physical address of the 4-KByte aligned PML4 table used for linear-address translation[2]
60:M	Reserved (must be 0)
61	Enables LAM57 for user pointers; Section 4.4.[3]
62	Enables LAM48 for user pointers; ignored if bit 61 is set.[3]
63	Reserved (must be 0)[4]

NOTES:

1. Section 5.9.2 explains how the processor determines the memory type used to access the PML4 table during linear-address translation with CR4.PCIDE = 1.
2. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 5.1.4.
3. LAM is not a paging feature.
4. See Section 5.10.4.1 for use of bit 63 of the source operand of the MOV to CR3 instruction.

After software modifies the value of CR4.PCIDE, the logical processor immediately begins using CR3 as specified for the new value. For example, if software changes CR4.PCIDE from 1 to 0, the current PCID immediately changes from CR3[11:0] to 000H (see also Section 5.10.4.1). In addition, the logical processor subsequently determines the memory type used to access the PML4 table using CR3.PWT and CR3.PCD, which had been bits 4:3 of the PCID.

5.5.3 Use of HLATP with HLAT 4-Level Paging and 5-Level Paging

With HLAT paging, 4-level paging and 5-level paging each translate linear addresses using a hierarchy of in-memory paging structures located using the value of HLATP (a VM-execution control field in the VMCS), which is used to locate the first paging structure. For 4-level paging, this is the PML4 table, and for 5-level paging it is the PML5 table.

HLATP has the same format as that given for CR3 in Table 5-12, with the exception that bits 2:0 and bits 11:5 are reserved and must be zero (these bits are checked by VM entry). HLATP does not contain a PCID value. HLAT paging with CR4.PCIDE = 1 uses the PCID value in CR3[11:0].

5.5.4 Linear-Address Translation with 4-Level Paging and 5-Level Paging

4-level paging and 5-level paging may map linear addresses to 4-KByte pages, 2-MByte pages, or 1-GByte pages.[1] Figure 5-8 illustrates the translation process for 4-level paging when it produces a 4-KByte page; Figure 5-9 covers the case of a 2-MByte page, and Figure 5-10 the case of a 1-GByte page. (The process for 5-level paging is similar.)

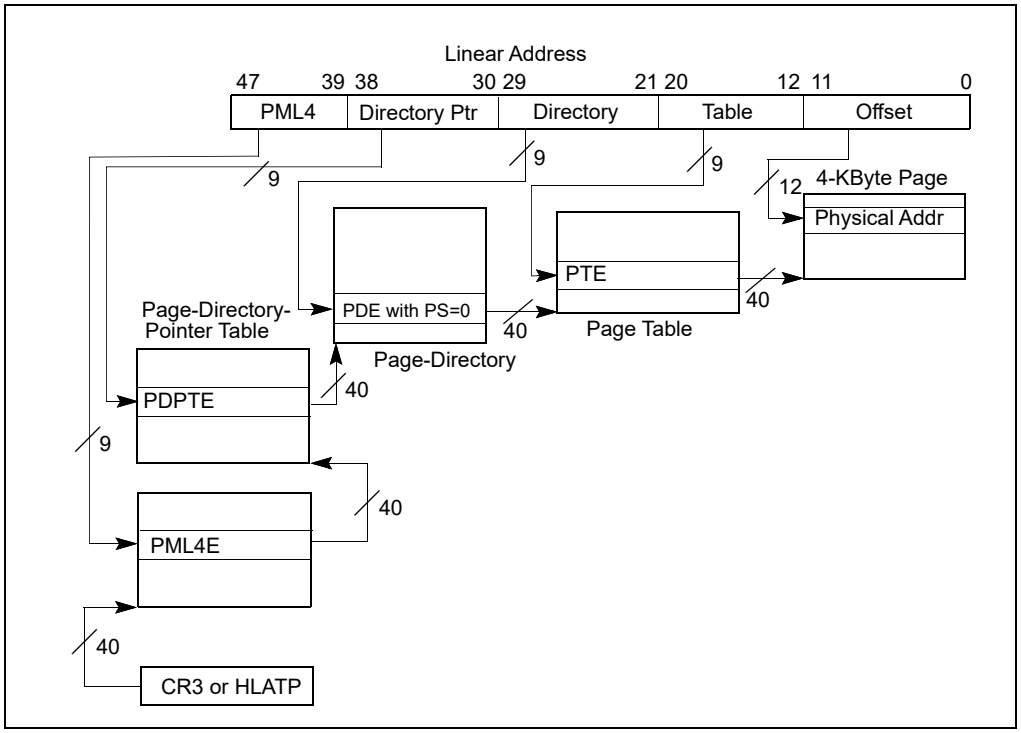


Figure 5-8. Linear-Address Translation to a 4-KByte Page Using 4-Level Paging

1. Not all processors support 1-GByte pages; see Section 5.1.4.

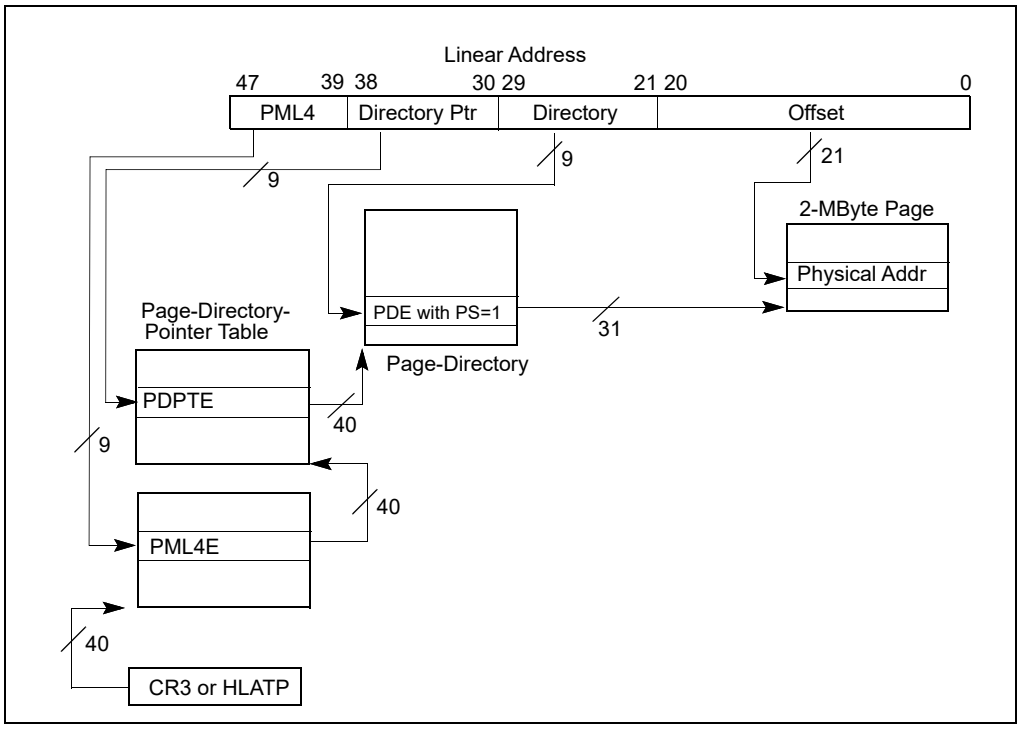


Figure 5-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

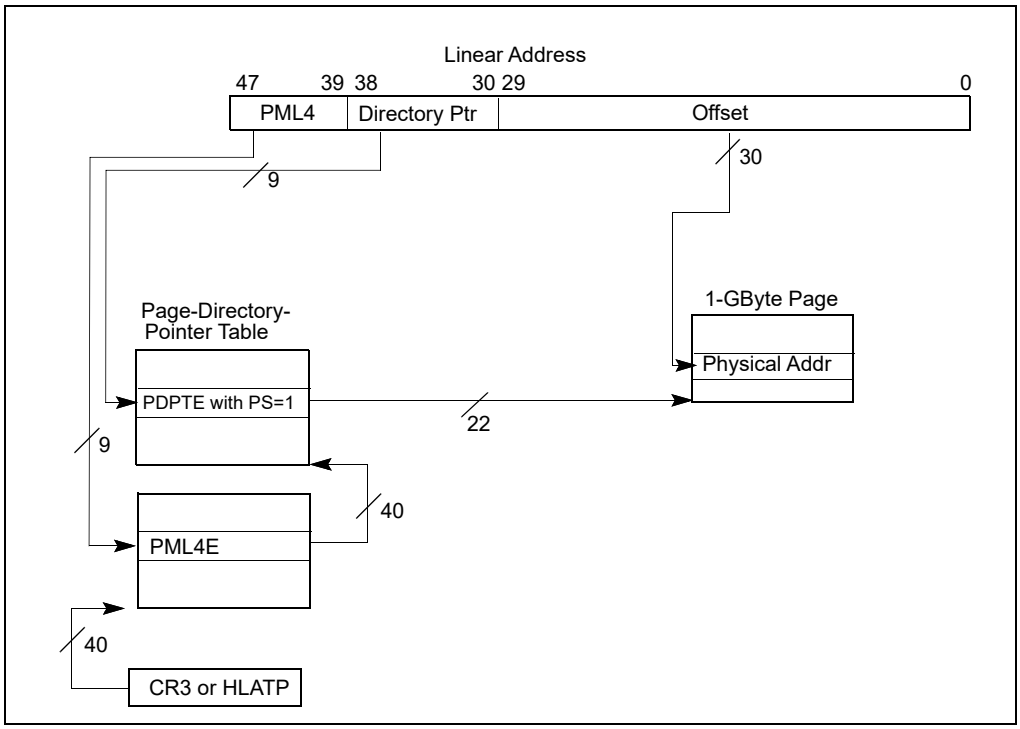


Figure 5-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

4-level paging and 5-level paging associate with each linear address a **protection key**. Section 5.6 explains how the processor uses the protection key in its determination of the access rights of each linear address.

The remainder of this section describes the translation process used by 4-level paging and 5-level paging in more detail, as well has how the page size and protection key are determined. Because the process used by the two paging modes is similar, they are described together, with any differences identified, in the following items:

- With 5-level paging, a 4-KByte naturally aligned PML5 table is located at the physical address specified in bits 51:12 of CR3 (see Table 5-12). (4-level paging does not use a PML5 table and omits this step.) A PML5 table comprises 512 64-bit entries (PML5Es). A PML5E is selected using the physical address defined as follows:
  - Bits 51:12 are from CR3 or HLATP.
  - Bits 11:3 are bits 56:48 of the linear address.
  - Bits 2:0 are all 0.

  Because a PML5E is identified using bits 56:48 of the linear address, it controls access to a 256-TByte region of the linear-address space.

  With HLAT paging, if bit 11 of the PML5E is 1, translation is restarted with ordinary paging with a maximum page size of 256-TBytes (see Section 5.5.5). Otherwise, the translation process continues as described in the next item.
- A 4-KByte naturally aligned PML4 table is located at the physical address specified in bits 51:12 of CR3 (for 4-level paging; see Table 5-12) or in bits 51:12 of the PML5E (for 5-level paging; see Table 5-14). A PML4 table comprises 512 64-bit entries (PML4Es). A PML4E is selected using the physical address defined as follows:
  - Bits 51:12 are from CR3 or the HLATP (for 4-level paging) or in bits 51:12 of the PML5E (for 5-level paging).
  - Bits 11:3 are bits 47:39 of the linear address.
  - Bits 2:0 are all 0.

  Because a PML4E is identified using bits 47:39 of the linear address, it controls access to a 512-GByte region of the linear-address space.

  With HLAT paging, if bit 11 of the PML4E is 1, translation is restarted with ordinary paging with a maximum page size of 512-GBytes (see Section 5.5.5). Otherwise, the translation process continues as described in the next item.
- A 4-KByte naturally aligned page-directory-pointer table is located at the physical address specified in bits 51:12 of the PML4E (see Table 5-15). A page-directory-pointer table comprises 512 64-bit entries (PDPTEs). A PDPTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PML4E.
  - Bits 11:3 are bits 38:30 of the linear address.
  - Bits 2:0 are all 0.

Because a PDPTE is identified using bits 47:30 of the linear address, it controls access to a 1-GByte region of the linear-address space.

With HLAT paging, if bit 11 of the PDPTE is 1, translation is restarted with ordinary paging with a maximum page size of 1-GByte (see Section 5.5.5). Otherwise, the translation process continues as described below.

Use of the PDPTE depends on its PS flag (bit 7):[1]

- If the PDPTE's PS flag is 1, the PDPTE maps a 1-GByte page (see Table 5-16). The final physical address is computed as follows:
  - Bits 51:30 are from the PDPTE.
  - Bits 29:0 are from the original linear address.

  The linear address's protection key is the value of bits 62:59 of the PDPTE (see Section 5.6.2).

1. The PS flag of a PDPTE is reserved and must be 0 (if the P flag is 1) if 1-GByte pages are not supported. See Section 5.1.4 for how to determine whether 1-GByte pages are supported.

- If the PDPTE's PS flag is 0, a 4-KByte naturally aligned page directory is located at the physical address specified in bits 51:12 of the PDPTE (see Table 5-17). A page directory comprises 512 64-bit entries (PDEs). A PDE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PDPTE.
  - Bits 11:3 are bits 29:21 of the linear address.
  - Bits 2:0 are all 0.

Because a PDE is identified using bits 47:21 of the linear address, it controls access to a 2-MByte region of the linear-address space.

With HLAT paging, if bit 11 of the PDE is 1, translation is restarted with ordinary paging with a maximum page size of 2-MBytes (see Section 5.5.5). Otherwise, the translation process continues as described below.

Use of the PDE depends on its PS flag:

- If the PDE's PS flag is 1, the PDE maps a 2-MByte page (see Table 5-18). The final physical address is computed as follows:
  - Bits 51:21 are from the PDE.
  - Bits 20:0 are from the original linear address.

  The linear address's protection key is the value of bits 62:59 of the PDE (see Section 5.6.2).
- If the PDE's PS flag is 0, a 4-KByte naturally aligned page table is located at the physical address specified in bits 51:12 of the PDE (see Table 5-19). A page table comprises 512 64-bit entries (PTEs). A PTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PDE.
  - Bits 11:3 are bits 20:12 of the linear address.
  - Bits 2:0 are all 0.
- Because a PTE is identified using bits 47:12 of the linear address, every PTE maps a 4-KByte page (see Table 5-20).

  With HLAT paging, if bit 11 of the PTE is 1, translation is restarted with ordinary paging with a maximum page size of 4-KBytes (see Section 5.5.5). Otherwise, the final physical address is computed as follows:
  - Bits 51:12 are from the PTE.
  - Bits 11:0 are from the original linear address.

  The linear address's protection key is the value of bits 62:59 of the PTE (see Section 5.6.2).

If a paging-structure entry's P flag (bit 0) is 0 or if the entry sets any reserved bit, the entry is used neither to reference another paging-structure entry nor to map a page. There is no translation for a linear address whose translation would use such a paging-structure entry; a reference to such a linear address causes a page-fault exception (see Section 5.7).

The following bits in a paging-structure entry are reserved with 4-level paging and 5-level paging (assuming that the entry's P flag is 1):

- Bits 51:MAXPHYADDR are reserved in every paging-structure entry.
- The PS flag is reserved in a PML5E or a PML4E.
- If 1-GByte pages are not supported, the PS flag is reserved in a PDPTE.[1]
- If the PS flag in a PDPTE is 1, bits 29:13 of the entry are reserved.
- If the PS flag in a PDE is 1, bits 20:13 of the entry are reserved.
- If IA32_EFER.NXE = 0, the XD flag (bit 63) is reserved in every paging-structure entry.

A reference using a linear address that is successfully translated to a physical address is performed only if allowed by the access rights of the translation; see Section 5.6.

1. See Section 5.1.4 for how to determine whether 1-GByte pages are supported.

Figure 5-11 gives a summary of the formats of CR3 and the 4-level and 5-level paging-structure entries. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are "not present"; bit 0 (P) and bit 7 (PS) are highlighted because they determine how a paging-structure entry is used.

Table 5-14. Format of a PML5 Entry (PML5E) that References a PML4 Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a PML4 table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 256-TByte region controlled by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 256-TByte region controlled by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the PML4 table referenced by this entry (see Section 5.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the PML4 table referenced by this entry (see Section 5.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 5.8)
6	Ignored
7 (PS)	Reserved (must be 0)
10:8	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
M–1:12	Physical address of 4-KByte aligned PML4 table referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 256-TByte region controlled by this entry; see Section 5.6); otherwise, reserved (must be 0)

Table 5-15. Format of a PML4 Entry (PML4E) that References a Page-Directory-Pointer Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page-directory-pointer table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 512-GByte region controlled by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 512-GByte region controlled by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page-directory-pointer table referenced by this entry (see Section 5.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page-directory-pointer table referenced by this entry (see Section 5.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 5.8)
6	Ignored

Table 5-15. Format of a PML4 Entry (PML4E) that References a Page-Directory-Pointer Table (Contd.)

Bit Position(s)	Contents
7 (PS)	Reserved (must be 0)
10:8	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
M–1:12	Physical address of 4-KByte aligned page-directory-pointer table referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 512-GByte region controlled by this entry; see Section 5.6); otherwise, reserved (must be 0)

Table 5-16. Format of a Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 1-GByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 1-GByte page referenced by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 1-GByte page referenced by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 1-GByte page referenced by this entry (see Section 5.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 1-GByte page referenced by this entry (see Section 5.9.2)
5 (A)	Accessed; indicates whether software has accessed the 1-GByte page referenced by this entry (see Section 5.8)
6 (D)	Dirty; indicates whether software has written to the 1-GByte page referenced by this entry (see Section 5.8)
7 (PS)	Page size; must be 1 (otherwise, this entry references a page directory; see Table 5-17)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 5.10); ignored otherwise
10:9	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
12 (PAT)	Indirectly determines the memory type used to access the 1-GByte page referenced by this entry (see Section 5.9.2)[1]
29:13	Reserved (must be 0)
(M–1):30	Physical address of the 1-GByte page referenced by this entry

Table 5-16. Format of a Page-Directory-Pointer-Table Entry (PDPTE) that Maps a 1-GByte Page (Contd.)

Bit Position(s)	Contents
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1 or CR4.PKS = 1, this may control the page's access rights (see Section 5.6.2); otherwise, it is ignored and not used to control access rights.
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 1-GByte page controlled by this entry; see Section 5.6); otherwise, reserved (must be 0)

NOTES:

1. The PAT is supported on all processors that support 4-level paging.

Table 5-17. Format of a Page-Directory-Pointer-Table Entry (PDPTE) that References a Page Directory

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page directory
1 (R/W)	Read/write; if 0, writes may not be allowed to the 1-GByte region controlled by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 1-GByte region controlled by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 5.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 5.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 5.8)
6	Ignored
7 (PS)	Page size; must be 0 (otherwise, this entry maps a 1-GByte page; see Table 5-16)
10:8	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
(M–1):12	Physical address of 4-KByte aligned page directory referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 1-GByte region controlled by this entry; see Section 5.6); otherwise, reserved (must be 0)

Table 5-18. Format of a Page-Directory Entry that Maps a 2-MByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 2-MByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 2-MByte page referenced by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte page referenced by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 5.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 5.9.2)
5 (A)	Accessed; indicates whether software has accessed the 2-MByte page referenced by this entry (see Section 5.8)
6 (D)	Dirty; indicates whether software has written to the 2-MByte page referenced by this entry (see Section 5.8)
7 (PS)	Page size; must be 1 (otherwise, this entry references a page table; see Table 5-19)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 5.10); ignored otherwise
10:9	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
12 (PAT)	Indirectly determines the memory type used to access the 2-MByte page referenced by this entry (see Section 5.9.2)
20:13	Reserved (must be 0)
(M–1):21	Physical address of the 2-MByte page referenced by this entry
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1 or CR4.PKS = 1, this may control the page's access rights (see Section 5.6.2); otherwise, it is ignored and not used to control access rights.
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte page controlled by this entry; see Section 5.6); otherwise, reserved (must be 0)

Table 5-19. Format of a Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 2-MByte region controlled by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte region controlled by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 5.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 5.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 5.8)
6	Ignored
7 (PS)	Page size; must be 0 (otherwise, this entry maps a 2-MByte page; see Table 5-18)
10:8	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
(M–1):12	Physical address of 4-KByte aligned page table referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte region controlled by this entry; see Section 5.6); otherwise, reserved (must be 0)

Table 5-20. Format of a Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 5.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 5.8)
7 (PAT)	Indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 5.10); ignored otherwise
10:9	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
(M–1):12	Physical address of the 4-KByte page referenced by this entry
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1 or CR4.PKS = 1, this may control the page's access rights (see Section 5.6.2); otherwise, it is ignored and not used to control access rights.
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 5.6); otherwise, reserved (must be 0)

63	62	61	60	59	58	57	56	55	54	53	52	51		M[1]	M-1			32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved[2]															Address of PML4 table (4-level paging) or PML5 table (5-level paging)																									Ignored						PCD	PWT	Ign.			**CR3**
XD[3]	Ignored												Rsvd.		Address of PML4 table																									R[4]	Ign.			Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	**PML5E: present**
Ignored																																																		0	**PML5E: not present**
XD	Ignored												Rsvd.		Address of page-directory-pointer table																									R	Ign.			Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	**PML4E: present**
Ignored																																																		0	**PML4E: not present**
XD	Prot. Key[5]		Ignored										Rsvd.		Address of 1GB page frame												Reserved											PAT	R	Ign.		G	1	D	A	PCD	PWT	U/S	R/W	1	**PDPTE: 1GB page**
XD	Ignored												Rsvd.		Address of page directory																									R	Ign.			0	Ign	A	PCD	PWT	U/S	R/W	1	**PDPTE: page directory**
Ignored																																																		0	**PDTPE: not present**
XD	Prot. Key		Ignored										Rsvd.		Address of 2MB page frame																		Reserved					PAT	R	Ign.		G	1	D	A	PCD	PWT	U/S	R/W	1	**PDE: 2MB page**
XD	Ignored												Rsvd.		Address of page table																									R	Ign.			0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**
Ignored																																																		0	**PDE: not present**
XD	Prot. Key		Ignored										Rsvd.		Address of 4KB page frame																									R	Ign.		G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**
Ignored																																																		0	**PTE: not present**

Figure 5-11. Formats of CR3 and Paging-Structure Entries with 4-Level Paging and 5-Level Paging

NOTES:

1. M is an abbreviation for MAXPHYADDR.
2. Reserved fields must be 0. On processors that support linear-address masking (see Section 4.4), bits 62:61 configure that feature and may be set to 1. Because linear-address masking is not a paging feature, those bits are not illustrated here.
3. If IA32_EFER.NXE = 0 and the P flag of a paging-structure entry is 1, the XD flag (bit 63) is reserved.
4. Bit 11 is R (restart) only for HLAT paging; it is ignored for ordinary paging.
5. The protection key is used only if software has enabled the appropriate feature; see Section 5.6.2. It is ignored otherwise.

5.5.5 Restart of HLAT Paging

As noted in Section 5.5.1, HLAT paging may specify that a translation of a linear address must be restarted. Specifically, this occurs when HLAT paging encounters a paging-structure entry that sets bit 11 (see Section 5.5.4).

When this occurs, translation of the linear address is restarted using ordinary paging (starting with a paging structure identified using CR3). The restarted translation proceeds just as if the HLAT feature were not enabled. The entire linear address is translated again, including those portions that had been used by HLAT paging prior to the restart.

The process of restarting HLAT paging (using ordinary paging) always specifies a maximum page size to be used when a resulting translation is cached in the TLBs. This maximum page size depends on the level of the paging-structure entry that restarts the translation by setting bit 11; details are given in Section 5.5.4. The page size of the translation produced by the restarted process is never greater than this maximum page size. See Section 5.10.2.2 for more discussion.

5.6 ACCESS RIGHTS

There is a translation for a linear address if the processes described in Section 5.3, Section 5.4.2, and Section 5.5 (depending upon the paging mode) completes and produces a physical address. Whether an access is permitted by a translation is determined by the access rights specified by the paging-structure entries controlling the translation;[1] paging-mode modifiers in CR0, CR4, and the IA32_EFER MSR; EFLAGS.AC; and the mode of the access.

Section 5.6.1 describes how the processor determines the access rights for each linear address. Section 5.6.2 provides additional information about how protection keys contribute to access-rights determination. (They do so only with 4-level paging and 5-level paging, and only if CR4.PKE = 1 or CR4.PKS = 1.)

NOTE

If HLAT paging is restarted, permissions are determined only by the access rights specified by the paging-structure entries that the subsequent ordinary paging used to translate the linear address. The access rights specified by the entries used earlier by HLAT paging do not apply.

5.6.1 Determination of Access Rights

Every access to a linear address is either a **supervisor-mode access** or a **user-mode access**. For all instruction fetches and most data accesses, this distinction is determined by the current privilege level (CPL): accesses made while CPL < 3 are supervisor-mode accesses, while accesses made while CPL = 3 are user-mode accesses.

Some operations implicitly access system data structures with linear addresses; the resulting accesses to those data structures are supervisor-mode accesses regardless of CPL. Examples of such accesses include the following: accesses to the global descriptor table (GDT) or local descriptor table (LDT) to load a segment descriptor; accesses to the interrupt descriptor table (IDT) when delivering an interrupt or exception; and accesses to the task-state segment (TSS) as part of a task switch or change of CPL. All these accesses are called **implicit supervisor-mode accesses** regardless of CPL. Other accesses made while CPL < 3 are called **explicit supervisor-mode accesses**.

Access rights are also controlled by the **mode** of a linear address as specified by the paging-structure entries controlling the translation of the linear address. If the U/S flag (bit 2) is 0 in at least one of the paging-structure entries, the address is a **supervisor-mode address**. Otherwise, the address is a **user-mode address**.

When the shadow-stack feature of control-flow enforcement technology (CET) is enabled, certain accesses to linear addresses are considered **shadow-stack accesses** (see Section 18.2, "Shadow Stacks," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). Like ordinary data accesses, each shadow-stack access is defined as being either a user access or a supervisor access. In general, a shadow-stack access is a user access if CPL = 3 and a supervisor access if CPL < 3. The WRUSS instruction is an exception; although it can be executed only if CPL = 0, the processor treats its shadow-stack accesses as user accesses.

1. With PAE paging, the PDPTEs do not determine access rights.

Shadow-stack accesses are allowed only to **shadow-stack addresses**. A linear address is a shadow-stack address if the following are true of the translation of the linear address: (1) the R/W flag (bit 1) is 0 and the dirty flag (bit 6) is 1 in the paging-structure entry that maps the page containing the linear address; and (2) the R/W flag is 1 in every other paging-structure entry controlling the translation of the linear address.

The following items detail how paging determines access rights (only the items noted explicitly apply to shadow-stack accesses):

NOTE

Many of the items below refer to an address with a protection key for which read (or write) access is permitted. Section 5.6.2 provides details on when a protection key will permit (or not permit) a data access (read or write) to a linear address using that protection key.

- For supervisor-mode accesses:
  - Data may be read (implicitly or explicitly) from any supervisor-mode address with a protection key for which read access is permitted.
  - Data reads from user-mode pages.
    Access rights depend on the value of CR4.SMAP:
    - If CR4.SMAP = 0, data may be read from any user-mode address with a protection key for which read access is permitted.
    - If CR4.SMAP = 1, access rights depend on the value of EFLAGS.AC and whether the access is implicit or explicit:
      - If EFLAGS.AC = 1 and the access is explicit, data may be read from any user-mode address with a protection key for which read access is permitted.
      - If EFLAGS.AC = 0 or the access is implicit, data may not be read from any user-mode address.
  - Data writes to supervisor-mode addresses.
    Access rights depend on the value of CR0.WP:
    - If CR0.WP = 0, data may be written to any supervisor-mode address with a protection key for which write access is permitted.
    - If CR0.WP = 1, data may be written to any supervisor-mode address with a translation for which the R/W flag (bit 1) is 1 in every paging-structure entry controlling the translation and with a protection key for which write access is permitted; data may not be written to any supervisor-mode address with a translation for which the R/W flag is 0 in any paging-structure entry controlling the translation.
  - Data writes to user-mode addresses.
    Access rights depend on the value of CR0.WP:
    - If CR0.WP = 0, access rights depend on the value of CR4.SMAP:
      - If CR4.SMAP = 0, data may be written to any user-mode address with a protection key for which write access is permitted.
      - If CR4.SMAP = 1, access rights depend on the value of EFLAGS.AC and whether the access is implicit or explicit:
        - If EFLAGS.AC = 1 and the access is explicit, data may be written to any user-mode address with a protection key for which write access is permitted.
        - If EFLAGS.AC = 0 or the access is implicit, data may not be written to any user-mode address.
    - If CR0.WP = 1, access rights depend on the value of CR4.SMAP:
      - If CR4.SMAP = 0, data may be written to any user-mode address with a translation for which the R/W flag is 1 in every paging-structure entry controlling the translation and with a protection key for which write access is permitted; data may not be written to any user-mode address with a translation for which the R/W flag is 0 in any paging-structure entry controlling the translation.
      - If CR4.SMAP = 1, access rights depend on the value of EFLAGS.AC and whether the access is implicit or explicit:

        - If EFLAGS.AC = 1 and the access is explicit, data may be written to any user-mode address with a translation for which the R/W flag is 1 in every paging-structure entry controlling the translation and with a protection key for which write access is permitted; data may not be written to any user-mode address with a translation for which the R/W flag is 0 in any paging-structure entry controlling the translation.
        - If EFLAGS.AC = 0 or the access is implicit, data may not be written to any user-mode address.
    - Instruction fetches from supervisor-mode addresses.
        - For 32-bit paging or if IA32_EFER.NXE = 0, instructions may be fetched from any supervisor-mode address.
        - For other paging modes with IA32_EFER.NXE = 1, instructions may be fetched from any supervisor-mode address with a translation for which the XD flag (bit 63) is 0 in every paging-structure entry controlling the translation; instructions may not be fetched from any supervisor-mode address with a translation for which the XD flag is 1 in any paging-structure entry controlling the translation.
    - Instruction fetches from user-mode addresses.
      Access rights depend on the values of CR4.SMEP:
        - If CR4.SMEP = 0, access rights depend on the paging mode and the value of IA32_EFER.NXE:
            - For 32-bit paging or if IA32_EFER.NXE = 0, instructions may be fetched from any user-mode address.
            - For other paging modes with IA32_EFER.NXE = 1, instructions may be fetched from any user-mode address with a translation for which the XD flag is 0 in every paging-structure entry controlling the translation; instructions may not be fetched from any user-mode address with a translation for which the XD flag is 1 in any paging-structure entry controlling the translation.
        - If CR4.SMEP = 1, instructions may not be fetched from any user-mode address.
    - Supervisor-mode shadow-stack accesses are allowed only to supervisor-mode shadow-stack addresses (see above).
- For user-mode accesses:
    - Data reads.
      Access rights depend on the mode of the linear address:
        - Data may be read from any user-mode address with a protection key for which read access is permitted.
        - Data may not be read from any supervisor-mode address.
    - Data writes.
      Access rights depend on the mode of the linear address:
        - Data may be written to any user-mode address with a translation for which the R/W flag is 1 in every paging-structure entry controlling the translation and with a protection key for which write access is permitted.
        - Data may not be written to any supervisor-mode address.
    - Instruction fetches.
      Access rights depend on the mode of the linear address, the paging mode, and the value of IA32_EFER.NXE:
        - For 32-bit paging or if IA32_EFER.NXE = 0, instructions may be fetched from any user-mode address.
        - For other paging modes with IA32_EFER.NXE = 1, instructions may be fetched from any user-mode address with a translation for which the XD flag is 0 in every paging-structure entry controlling the translation.
        - Instructions may not be fetched from any supervisor-mode address.
    - User-mode shadow-stack accesses made outside enclave mode are allowed only to user-mode shadow-stack addresses (see above). User-mode shadow-stack accesses made in enclave mode are treated like ordinary data accesses (see above).

A processor may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 5.10). These structures may include information about access rights. The processor may enforce access rights based on the TLBs and paging-structure caches instead of on the paging structures in memory.

This fact implies that, if software modifies a paging-structure entry to change access rights, the processor might not use that change for a subsequent access to an affected linear address (see Section 5.10.4.3). See Section 5.10.4.2 for how software can ensure that the processor uses the modified access rights.

5.6.2 Protection Keys

4-level paging and 5-level paging associate a 4-bit protection key with each linear address (the protection key located in bits 62:59 of the paging-structure entry that mapped the page containing the linear address; see Section 5.5). Two protection key features control accesses to linear addresses based on their protection keys:

- If CR4.PKE = 1, the PKRU register determines, for each protection key, whether user-mode addresses with that protection key may be read or written.
- If CR4.PKS = 1, the IA32_PKRS MSR (MSR index 6E1H) determines, for each protection key, whether supervisor-mode addresses with that protection key may be read or written.

32-bit paging and PAE paging do not associate linear addresses with protection keys. For the purposes of Section 5.6.1, reads and writes are implicitly permitted for all protection keys with either of those paging modes.

The PKRU register (protection-key rights for user pages) is a 32-bit register with the following format: for each *i* (0 ? *i* ? 15), PKRU[2*i*] is the **access-disable bit** for protection key *i* (AD*i*); PKRU[2*i*+1] is the **write-disable bit** for protection key *i* (WD*i*). The IA32_PKRS MSR has the same format (bits 63:32 of the MSR are reserved and must be zero).

Software can use the RDPKRU and WRPKRU instructions with ECX = 0 to read and write PKRU. In addition, the PKRU register is XSAVE-managed state and can thus be read and written by instructions in the XSAVE feature set. See Chapter 13, "Managing State Using the XSAVE Feature Set," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information about the XSAVE feature set.

Software can use the RDMSR and WRMSR instructions to read and write the IA32_PKRS MSR. Writes to the IA32_PKRS MSR using WRMSR are not serializing. The IA32_PKRS MSR is not XSAVE-managed.

How a linear address's protection key controls access to the address depends on the mode of the linear address:

- A linear address's protection key controls only data accesses to the address. It does not in any way affect instructions fetches from the address.
- If CR4.PKE = 0, the protection key of a user-mode address does not control data accesses to the address (for the purposes of Section 5.6.1, reads and writes of user-mode addresses are implicitly permitted for all protection keys).

  If CR4.PKE = 1, use of the protection key *i* of a user-mode address depends on the value of the PKRU register:

  — If AD*i* = 1, no data accesses are permitted.

  — If WD*i* = 1, permission may be denied to certain data write accesses:

    - User-mode write accesses are not permitted.
    - Supervisor-mode write accesses are not permitted if CR0.WP = 1. (If CR0.WP = 0, WD*i* does not affect supervisor-mode write accesses to user-mode addresses with protection key *i*.)
- If CR4.PKS = 0, the protection key of a supervisor-mode address does not control data accesses to the address (for the purposes of Section 5.6.1, reads and writes of supervisor-mode addresses are implicitly permitted for all protection keys).

  If CR4.PKS = 1, use of the protection key *i* of a supervisor-mode address depends on the value of the IA32_PKRS MSR:

  — If AD*i* = 1, no data accesses are permitted.

  — If WD*i* = 1, write accesses are not permitted if CR0.WP = 1. (If CR0.WP = 0, IA32_PKRS.WD*i* does not affect write accesses to supervisor-mode addresses with protection key *i*.)

Protection keys apply to shadow-stack accesses just as they do to ordinary data accesses.

5.7 PAGE-FAULT EXCEPTIONS

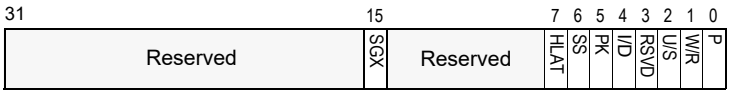
Accesses using linear addresses may cause **page-fault exceptions** (#PF; exception 14). An access to a linear address may cause a page-fault exception for either of two reasons: (1) there is no translation for the linear address; or (2) there is a translation for the linear address, but its access rights do not permit the access.

As noted in Section 5.3, Section 5.4.2, and Section 5.5, there is no translation for a linear address if the translation process for that address would use a paging-structure entry in which the P flag (bit 0) is 0 or one that sets a reserved bit.[1] If there is a translation for a linear address, its access rights are determined as specified in Section 5.6.

When Intel® Software Guard Extensions (Intel® SGX) are enabled, the processor may deliver exception 14 for reasons unrelated to paging. See Section 36.3, "Access-control Requirements," and Section 36.20, "Enclave Page Cache Map (EPCM)," in Chapter 36, "Enclave Access Control and Data Structures." Such an exception is called an **SGX-induced page fault**. The processor uses the error code to distinguish SGX-induced page faults from ordinary page faults.

When a page fault occurs, the processor loads the CR2 register with the linear address that generated the exception. If linear-address masking had been in effect (Section 4.4), the address recorded reflects the result of that masking and does not contain any masked metadata. 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 address are cleared.

Figure 5-12 illustrates the error code that the processor provides on delivery of a page-fault exception. The following items explain how the bits in the error code describe the nature of the page-fault exception:



P
0 The fault was caused by a non-present page.
1 The fault was caused by a page-level protection violation.

W/R
0 The access causing the fault was a read.
1 The access causing the fault was a write.

U/S
0 A supervisor-mode access caused the fault.
1 A user-mode access caused the fault.

RSVD
0 The fault was not caused by reserved bit violation.
1 The fault was caused by a reserved bit set to 1 in some paging-structure entry.

I/D
0 The fault was not caused by an instruction fetch.
1 The fault was caused by an instruction fetch.

PK
0 The fault was not caused by protection keys.
1 There was a protection-key violation.

SS
0 The fault was not caused by a shadow-stack access.
1 The fault was caused by a shadow-stack access.

HLAT
0 The fault occurred during ordinary paging or due to access rights.
1 The fault occurred during HLAT paging.

SGX
0 The fault is not related to SGX.
1 The fault resulted from violation of SGX-specific access-control requirements.

Figure 5-12. Page-Fault Error Code

1. If HLAT paging encounters a paging-structure entry that sets a reserved bit, there is no translation even if the bit 11 of the entry indicates a restart. In this case, there is a page fault and the translation is not restarted.

- P flag (bit 0).
  This flag is 0 if there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address.
- W/R (bit 1).
  If the access causing the page-fault exception was a write, this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
- U/S (bit 2).
  If a user-mode access caused the page-fault exception, this flag is 1; it is 0 if a supervisor-mode access did so. This flag describes the access causing the page-fault exception, not the access rights specified by paging. User-mode and supervisor-mode accesses are defined in Section 5.6.
- RSVD flag (bit 3).
  This flag is 1 if there is no translation for the linear address because a reserved bit was set in one of the paging-structure entries used to translate that address. (Because reserved bits are not checked in a paging-structure entry whose P flag is 0, bit 3 of the error code can be set only if bit 0 is also set.[1])

  Bits reserved in the paging-structure entries are reserved for future functionality. Software developers should be aware that such bits may be used in the future and that a paging-structure entry that causes a page-fault exception on one processor might not do so in the future.
- I/D flag (bit 4).
  This flag is 1 if (1) the access causing the page-fault exception was an instruction fetch; and (2) either (a) CR4.SMEP = 1; or (b) both (i) CR4.PAE = 1 (either PAE paging, 4-level paging, or 5-level paging is in use); and (ii) IA32_EFER.NXE = 1. Otherwise, the flag is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
- PK flag (bit 5).
  This flag is 1 only for data accesses and only with 4-level paging and 5-level paging. In these cases, the setting depends on the mode of the address being accessed:
  - For accesses to supervisor-mode addresses, the flag is set if (1) CR4.PKS = 1; (2) the linear address has protection key *i*; and (3) the IA32_PKRS MSR (see Section 5.6.2) is such that either (a) AD*i* = 1; or (b) the following all hold: (i) WD*i* = 1; (ii) the access is a write access; and (iii) either CR0.WP = 1 or the access causing the page-fault exception was a user-mode access. (Note that this flag may be set on page faults due to user-mode accesses to supervisor-mode addresses.)
  - For accesses to user-mode addresses, the flag is set if (1) CR4.PKE = 1; (2) the linear address has protection key *i*; and (3) the PKRU register (see Section 5.6.2) is such that either (a) AD*i* = 1; or (b) the following all hold: (i) WD*i* = 1; (ii) the access is a write access; and (iii) either CR0.WP = 1 or the access causing the page-fault exception was a user-mode access.
- SS (bit 6).
  If the access causing the page-fault exception was a shadow-stack access (including shadow-stack accesses in enclave mode), this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
- HLAT (bit 7).
  This flag is 1 if there is no translation for the linear address using HLAT paging because, in one of the paging-structure entries used to translate that address, either the P flag was 0 or a reserved bit was set. An error code will set this flag only if it clears bit 0 or sets bit 3. This flag will not be set by a page fault resulting from a violation of access rights, nor for one encountered during ordinary paging, including the case in which there has been a restart of HLAT paging.
- SGX flag (bit 15).
  This flag is 1 if the exception is unrelated to paging and resulted from violation of SGX-specific access-control requirements. Because such a violation can occur only if there is no ordinary page fault, this flag is set only if the P flag (bit 0) is 1 and the RSVD flag (bit 3) and the PK flag (bit 5) are both 0.

Page-fault exceptions occur only due to an attempt to use a linear address. Failures to load the PDPTE registers with PAE paging (see Section 5.4.1) cause general-protection exceptions (#GP(0)) and not page-fault exceptions.

1. Some past processors had errata for some page faults that occur when there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address. Due to these errata, some such page faults produced error codes that cleared bit 0 (P flag) and set bit 3 (RSVD flag).

5.8 ACCESSED AND DIRTY FLAGS

For any paging-structure entry that is used during linear-address translation, bit 5 is the **accessed** flag.[1] For paging-structure entries that map a page (as opposed to referencing another paging structure), bit 6 is the **dirty** flag. These flags are provided for use by memory-management software to manage the transfer of pages and paging structures into and out of physical memory.

Whenever the processor uses a paging-structure entry as part of linear-address translation, it sets the accessed flag in that entry (if it is not already set).

Whenever there is a write to a linear address, the processor sets the dirty flag (if it is not already set) in the paging-structure entry that identifies the final physical address for the linear address (either a PTE or a paging-structure entry in which the PS flag is 1).

The previous two paragraphs apply also to HLAT paging. If HLAT paging encounters a paging-structure entry that sets bit 11, indicating a restart, the processor will set the accessed flag in that entry; it will not set the dirty flag because, if an entry indicates a restart, it does identify the final physical address for the linear address being translated.

NOTE

If software on one logical processor writes to a page while software on another logical processor concurrently clears the R/W flag in the paging-structure entry that maps the page, execution on some processors may result in the entry's dirty flag being set (due to the write on the first logical processor) and the entry's R/W flag being clear (due to the update to the entry on the second logical processor). This will never occur on a processor that supports control-flow enforcement technology (CET). Specifically, a processor that supports CET will never set the dirty flag in a paging-structure entry in which the R/W flag is clear.

Memory-management software may clear these flags when a page or a paging structure is initially loaded into physical memory. These flags are "sticky," meaning that, once set, the processor does not clear them; only software can clear them.

A processor may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 5.10). This fact implies that, if software changes an accessed flag or a dirty flag from 1 to 0, the processor might not set the corresponding bit in memory on a subsequent access using an affected linear address (see Section 5.10.4.3). See Section 5.10.4.2 for how software can ensure that these bits are updated as desired.

NOTE

The accesses used by the processor to set these flags may or may not be exposed to the processor's self-modifying code detection logic. If the processor is executing code from the same memory area that is being used for the paging structures, the setting of these flags may or may not result in an immediate change to the executing code stream.

5.9 PAGING AND MEMORY TYPING

The **memory type** of a memory access refers to the type of caching used for that access. Chapter 13, "Memory Cache Control," provides many details regarding memory typing in the Intel-64 and IA-32 architectures. This section describes how paging contributes to the determination of memory typing.

The way in which paging contributes to memory typing depends on whether the processor supports the **Page Attribute Table** (**PAT**; see Section 13.12).[2] Section 5.9.1 and Section 5.9.2 explain how paging contributes to memory typing depending on whether the PAT is supported.

---

1. With PAE paging, the PDPTEs are not used during linear-address translation but only to load the PDPTE registers for some executions of the MOV CR instruction (see Section 5.4.1). For this reason, the PDPTEs do not contain accessed flags with PAE paging.
2. The PAT is supported on Pentium III and more recent processor families. See Section 5.1.4 for how to determine whether the PAT is supported.

5.9.1 Paging and Memory Typing When the PAT is Not Supported (Pentium Pro and Pentium II Processors)

NOTE

The PAT is supported on all processors that support 4-level paging or 5-level paging. Thus, this section applies only to 32-bit paging and PAE paging.

If the PAT is not supported, paging contributes to memory typing in conjunction with the memory-type range registers (MTRRs) as specified in Table 13-6 in Section 13.5.2.1.

For any access to a physical address, the table combines the memory type specified for that physical address by the MTRRs with a PCD value and a PWT value. The latter two values are determined as follows:

- For an access to a PDE with 32-bit paging, the PCD and PWT values come from CR3.
- For an access to a PDE with PAE paging, the PCD and PWT values come from the relevant PDPTE register.
- For an access to a PTE, the PCD and PWT values come from the relevant PDE.
- For an access to the physical address that is the translation of a linear address, the PCD and PWT values come from the relevant PTE (if the translation uses a 4-KByte page) or the relevant PDE (otherwise).
- With PAE paging, the UC memory type is used when loading the PDPTEs (see Section 5.4.1).

5.9.2 Paging and Memory Typing When the PAT is Supported (Pentium III and More Recent Processor Families)

If the PAT is supported, paging contributes to memory typing in conjunction with the PAT and the memory-type range registers (MTRRs) as specified in Table 13-7 in Section 13.5.2.2.

The PAT is a 64-bit MSR (IA32_PAT; MSR index 277H) comprising eight (8) 8-bit entries (entry *i* comprises bits 8*i*+7:8*i* of the MSR).

For any access to a physical address, the table combines the memory type specified for that physical address by the MTRRs with a memory type selected from the PAT. Table 13-11 in Section 13.12.3 specifies how a memory type is selected from the PAT. Specifically, it comes from entry *i* of the PAT, where *i* is defined as follows:

- For an access to an entry in a paging structure whose address is in CR3 (e.g., the PML4 table with 4-level paging):
  - For 4-level paging or 5-level paging with CR4.PCIDE = 1, *i* = 0.
  - Otherwise, *i* = 2*PCD+PWT, where the PCD and PWT values come from CR3.
- For an access to a PDE with PAE paging, *i* = 2*PCD+PWT, where the PCD and PWT values come from the relevant PDPTE register.
- For an access to a paging-structure entry X whose address is in another paging-structure entry Y, *i* = 2*PCD+PWT, where the PCD and PWT values come from Y.
- For an access to the physical address that is the translation of a linear address, *i* = 4*PAT+2*PCD+PWT, where the PAT, PCD, and PWT values come from the relevant PTE (if the translation uses a 4-KByte page), the relevant PDE (if the translation uses a 2-MByte page or a 4-MByte page), or the relevant PDPTE (if the translation uses a 1-GByte page).
- With PAE paging, the WB memory type is used when loading the PDPTEs (see Section 5.4.1).[1]

1. Some older IA-32 processors used the UC memory type when loading the PDPTEs. Some processors may use the UC memory type if CR0.CD = 1 or if the MTRRs are disabled. These behaviors are model-specific and not architectural.

5.9.3 Caching Paging-Related Information about Memory Typing

A processor may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 5.10). These structures may include information about memory typing. The processor may use memory-typing information from the TLBs and paging-structure caches instead of from the paging structures in memory.

This fact implies that, if software modifies a paging-structure entry to change the memory-typing bits, the processor might not use that change for a subsequent translation using that entry or for access to an affected linear address. See Section 5.10.4.2 for how software can ensure that the processor uses the modified memory typing.

5.10 CACHING TRANSLATION INFORMATION

The Intel-64 and IA-32 architectures may accelerate the address-translation process by caching data from the paging structures on the processor. Because the processor does not ensure that the data that it caches are always consistent with the structures in memory, it is important for software developers to understand how and when the processor may cache such data. They should also understand what actions software can take to remove cached data that may be inconsistent and when it should do so. This section provides software developers information about the relevant processor operation.

Section 5.10.1 introduces process-context identifiers (PCIDs), which a logical processor may use to distinguish information cached for different linear-address spaces. Section 5.10.2 and Section 5.10.3 describe how the processor may cache information in translation lookaside buffers (TLBs) and paging-structure caches, respectively. Section 5.10.4 explains how software can remove inconsistent cached information by invalidating portions of the TLBs and paging-structure caches. Section 5.10.5 describes special considerations for multiprocessor systems.

5.10.1 Process-Context Identifiers (PCIDs)

Process-context identifiers (**PCIDs**) are a facility by which a logical processor may cache information for multiple linear-address spaces. The processor may retain cached information when software switches to a different linear-address space with a different PCID (e.g., by loading CR3; see Section 5.10.4.1 for details).

A PCID is a 12-bit identifier. Non-zero PCIDs are enabled by setting the PCIDE flag (bit 17) of CR4. If CR4.PCIDE = 0, the current PCID is always 000H; otherwise, the current PCID is the value of bits 11:0 of CR3.[1] Not all processors allow CR4.PCIDE to be set to 1; see Section 5.1.4 for how to determine whether this is allowed.

The processor ensures that CR4.PCIDE can be 1 only in IA-32e mode (thus, 32-bit paging and PAE paging use only PCID 000H). In addition, software can change CR4.PCIDE from 0 to 1 only if CR3[11:0] = 000H. These requirements are enforced by the following limitations on the MOV CR instruction:

- MOV to CR4 causes a general-protection exception (#GP) if it would change CR4.PCIDE from 0 to 1 and either IA32_EFER.LMA = 0 or CR3[11:0] ≠ 000H.
- MOV to CR0 causes a general-protection exception if it would clear CR0.PG to 0 while CR4.PCIDE = 1.

When a logical processor creates entries in the TLBs (Section 5.10.2) and paging-structure caches (Section 5.10.3), it associates those entries with the current PCID. When using entries in the TLBs and paging-structure caches to translate a linear address, a logical processor uses only those entries associated with the current PCID (see Section 5.10.2.4 for an exception).

If CR4.PCIDE = 0, a logical processor does not cache information for any PCID other than 000H. This is because (1) if CR4.PCIDE = 0, the logical processor will associate any newly cached information with the current PCID, 000H; and (2) if MOV to CR4 clears CR4.PCIDE, all cached information is invalidated (see Section 5.10.4.1).

1. Note that, while HLAT paging (Section 5.5.3) does not use CR3 to locate the first paging structure, it does use the PCID in CR3[11:0] when CR4.PCIDE = 1.

NOTE

In revisions of this manual that were produced when no processors allowed CR4.PCIDE to be set to 1, Section 5.10, "Caching Translation Information," discussed the caching of translation information without any reference to PCIDs. While the section now refers to PCIDs in its specification of this caching, this documentation change is not intended to imply any change to the behavior of processors that do not allow CR4.PCIDE to be set to 1.

5.10.2 Translation Lookaside Buffers (TLBs)

A processor may cache information about the translation of linear addresses in translation lookaside buffers (TLBs). In general, TLBs contain entries that map page numbers to page frames; these terms are defined in Section 5.10.2.1. Section 5.10.2.2 describes how information may be cached in TLBs, and Section 5.10.2.3 gives details of TLB usage. Section 5.10.2.4 explains the global-page feature, which allows software to indicate that certain translations should receive special treatment when cached in the TLBs.

5.10.2.1 Page Numbers, Page Frames, and Page Offsets

Section 5.3, Section 5.4.2, and Section 5.5 give details of how the different paging modes translate linear addresses to physical addresses. Specifically, the upper bits of a linear address (called the **page number**) determine the upper bits of the physical address (called the **page frame**); the lower bits of the linear address (called the **page offset**) determine the lower bits of the physical address. The boundary between the page number and the page offset is determined by the **page size**. Specifically:

- 32-bit paging:
  - If the translation does not use a PTE (because CR4.PSE = 1 and the PS flag is 1 in the PDE used), the page size is 4 MBytes and the page number comprises bits 31:22 of the linear address.
  - If the translation does use a PTE, the page size is 4 KBytes and the page number comprises bits 31:12 of the linear address.
- PAE paging:
  - If the translation does not use a PTE (because the PS flag is 1 in the PDE used), the page size is 2 MBytes and the page number comprises bits 31:21 of the linear address.
  - If the translation does use a PTE, the page size is 4 KBytes and the page number comprises bits 31:12 of the linear address.
- 4-level paging and 5-level paging:
  - If the translation does not use a PDE (because the PS flag is 1 in the PDPTE used), the page size is 1 GByte and the page number comprises bits 47:30 of the linear address.
  - If the translation does use a PDE but does not uses a PTE (because the PS flag is 1 in the PDE used), the page size is 2 MBytes and the page number comprises bits 47:21 of the linear address.
  - If the translation does use a PTE, the page size is 4 KBytes and the page number comprises bits 47:12 of the linear address.
  - The page size identified by the preceding items may be reduced if there has been a restart of HLAT paging (see Section 5.5.5). Restart of HLAT paging always specifies a maximum page size; this page size is determined by the level of the paging-structure entry that caused the restart. The page size used by the translation is the minimum of the maximum page size specified by the restart and the page size determined by the restarted translation (as specified by the previous items).

    For example, suppose that HLAT paging encounters a PDE that sets bit 11, indicating a restart. As a result, the restart uses a maximum page size of 2 MBytes. Suppose that the restarted translation encounters a PDPTE that sets bit 7, indicating a 1-GByte page. In this case, the translation produced will have a page size of 2 MBytes (the smaller of the two sizes).

5.10.2.2 Caching Translations in TLBs

The processor may accelerate the paging process by caching individual translations in **translation lookaside buffers** (**TLBs**). Each entry in a TLB is an individual translation. Each translation is referenced by a page number. It contains the following information from the paging-structure entries used to translate linear addresses with the page number:

- The physical address corresponding to the page number (the page frame).
- The access rights from the paging-structure entries used to translate linear addresses with the page number (see Section 5.6):
  - The logical-AND of the R/W flags.
  - The logical-AND of the U/S flags.
  - The logical-OR of the XD flags (necessary only if IA32_EFER.NXE = 1).
  - The protection key (only with 4-level paging and 5-level paging).
- Attributes from a paging-structure entry that identifies the final page frame for the page number (either a PTE or a paging-structure entry in which the PS flag is 1):
  - The dirty flag (see Section 5.8).
  - The memory type (see Section 5.9).

(TLB entries may contain other information as well. A processor may implement multiple TLBs, and some of these may be for special purposes, e.g., only for instruction fetches. Such special-purpose TLBs may not contain some of this information if it is not necessary. For example, a TLB used only for instruction fetches need not contain information about the R/W and dirty flags.)

As noted in Section 5.10.1, any TLB entries created by a logical processor are associated with the current PCID.

Processors need not implement any TLBs. Processors that do implement TLBs may invalidate any TLB entry at any time. Software should not rely on the existence of TLBs or on the retention of TLB entries.

5.10.2.3 Details of TLB Use

Because the TLBs cache entries only for linear addresses with translations, there can be a TLB entry for a page number only if the P flag is 1 and the reserved bits are 0 in each of the paging-structure entries used to translate that page number. In addition, the processor does not cache a translation for a page number unless the accessed flag is 1 in each of the paging-structure entries used during translation; before caching a translation, the processor sets any of these accessed flags that is not already 1.

Subject to the limitations given in the previous paragraph, the processor may cache a translation for any linear address, even if that address is not used to access memory. For example, the processor may cache translations required for prefetches and for accesses that result from speculative execution that would never actually occur in the executed code path.

If the page number of a linear address corresponds to a TLB entry associated with the current PCID, the processor may use that TLB entry to determine the page frame, access rights, and other attributes for accesses to that linear address. In this case, the processor may not actually consult the paging structures in memory. The processor may retain a TLB entry unmodified even if software subsequently modifies the relevant paging-structure entries in memory. See Section 5.10.4.2 for how software can ensure that the processor uses the modified paging-structure entries.

If the paging structures specify a translation using a page larger than 4 KBytes, some processors may cache multiple smaller-page TLB entries for that translation. Each such TLB entry would be associated with a page number corresponding to the smaller page size (e.g., bits 47:12 of a linear address with 4-level paging), even though part of that page number (e.g., bits 20:12) is part of the offset with respect to the page specified by the paging structures. The upper bits of the physical address in such a TLB entry are derived from the physical address in the PDE used to create the translation, while the lower bits come from the linear address of the access for which the translation is created. There is no way for software to be aware that multiple translations for smaller pages have been used for a large page. For example, an execution of INVLPG for a linear address on such a page invalidates any and all smaller-page TLB entries for the translation of any linear address on that page.

If software modifies the paging structures so that the page size used for a 4-KByte range of linear addresses changes, the TLBs may subsequently contain multiple translations for the address range (one for each page size). A reference to a linear address in the address range may use any of these translations. Which translation is used may vary from one execution to another, and the choice may be implementation-specific.

5.10.2.4 Global Pages

The Intel-64 and IA-32 architectures also allow for **global pages** when the PGE flag (bit 7) is 1 in CR4. If the G flag (bit 8) is 1 in a paging-structure entry that maps a page (either a PTE or a paging-structure entry in which the PS flag is 1), any TLB entry cached for a linear address using that paging-structure entry is considered to be **global**. Because the G flag is used only in paging-structure entries that map a page, and because information from such entries is not cached in the paging-structure caches, the global-page feature does not affect the behavior of the paging-structure caches.

A logical processor may use a global TLB entry to translate a linear address, even if the TLB entry is associated with a PCID different from the current PCID.

5.10.3 Paging-Structure Caches

In addition to the TLBs, a processor may cache other information about the paging structures in memory.

5.10.3.1 Caches for Paging Structures

A processor may support any or all of the following paging-structure caches:

- **PML5E cache** (5-level paging only). Each PML5E-cache entry is referenced by a 9-bit value and is used for linear addresses for which bits 56:48 have that value. The entry contains information from the PML5E used to translate such linear addresses:
  - The physical address from the PML5E (the address of the PML4 table).
  - The value of the R/W flag of the PML5E.
  - The value of the U/S flag of the PML5E.
  - The value of the XD flag of the PML5E.
  - The values of the PCD and PWT flags of the PML5E.

  The following items detail how a processor may use the PML5E cache:
  - If the processor has a PML5E-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML5E in memory).
  - The processor does not create a PML5E-cache entry unless the P flag is 1 and all reserved bits are 0 in the PML5E in memory.
  - The processor does not create a PML5E-cache entry unless the accessed flag is 1 in the PML5E in memory; before caching a translation, the processor sets the accessed flag if it is not already 1.
  - The processor may create a PML5E-cache entry even if there are no translations for any linear address that might use that entry (e.g., because the P flags are 0 in all entries in the referenced PML4 table).
  - If the processor creates a PML5E-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML5E in memory.
- **PML4E cache** (4-level paging and 5-level paging only). The use of the PML4E cache depends on the paging mode:
  - For 4-level paging, each PML4E-cache entry is referenced by a 9-bit value and is used for linear addresses for which bits 47:39 have that value.
  - For 5-level paging, each PML4E-cache entry is referenced by an 18-bit value and is used for linear addresses for which bits 56:39 have that value.

  A PML4E-cache entry contains information from the PML5E and PML4E used to translate the relevant linear addresses (for 4-level paging, the PML5E does not apply):

— The physical address from the PML4E (the address of the page-directory-pointer table).

— The logical-AND of the R/W flags in the PML5E and the PML4E.

— The logical-AND of the U/S flags in the PML5E and the PML4E.

— The logical-OR of the XD flags in the PML5E and the PML4E.

— The values of the PCD and PWT flags of the PML4E.

The following items detail how a processor may use the PML4E cache:

— If the processor has a PML4E-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML5E and PML4E in memory).

— The processor does not create a PML4E-cache entry unless the P flags are 1 and all reserved bits are 0 in the PML5E and the PML4E in memory.

— The processor does not create a PML4E-cache entry unless the accessed flags are 1 in the PML5E and the PML4E in memory; before caching a translation, the processor sets any accessed flags that are not already 1.

— The processor may create a PML4E-cache entry even if there are no translations for any linear address that might use that entry (e.g., because the P flags are 0 in all entries in the referenced page-directory-pointer table).

— If the processor creates a PML4E-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML4E in memory.

- **PDPTE cache** (4-level paging and 5-level paging only).[1] The use of the PML4E cache depends on the paging mode:

  — For 4-level paging, each PDPTE-cache entry is referenced by an 18-bit value and is used for linear addresses for which bits 47:30 have that value.

  — For 5-level paging, each PDPTE-cache entry is referenced by a 27-bit value and is used for linear addresses for which bits 56:30 have that value.

  A PDPTE-cache entry contains information from the PML5E, PML4E, PDPTE used to translate the relevant linear addresses (for 4-level paging, the PML5E does not apply):

  — The physical address from the PDPTE (the address of the page directory). (No PDPTE-cache entry is created for a PDPTE that maps a 1-GByte page.)

  — The logical-AND of the R/W flags in the PML5E, PML4E, and PDPTE.

  — The logical-AND of the U/S flags in the PML5E, PML4E, and PDPTE.

  — The logical-OR of the XD flags in the PML5E, PML4E, and PDPTE.

  — The values of the PCD and PWT flags of the PDPTE.

  The following items detail how a processor may use the PDPTE cache:

  — If the processor has a PDPTE-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML5E, PML4E, and PDPTE in memory).

  — The processor does not create a PDPTE-cache entry unless the P flags are 1, the PS flags are 0, and the reserved bits are 0 in the PML5E, PML4E, and PDPTE in memory.

  — The processor does not create a PDPTE-cache entry unless the accessed flags are 1 in the PML5E, PML4E, and PDPTE in memory; before caching a translation, the processor sets any accessed flags that are not already 1.

  — The processor may create a PDPTE-cache entry even if there are no translations for any linear address that might use that entry.

  — If the processor creates a PDPTE-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML5E, PML4E, or PDPTE in memory.

1. With PAE paging, the PDPTEs are stored in internal, non-architectural registers. The operation of these registers is described in Section 5.4.1 and differs from that described here.

- **PDE cache**. The use of the PDE cache depends on the paging mode:
  - For 32-bit paging, each PDE-cache entry is referenced by a 10-bit value and is used for linear addresses for which bits 31:22 have that value.
  - For PAE paging, each PDE-cache entry is referenced by an 11-bit value and is used for linear addresses for which bits 31:21 have that value.
  - For 4-level paging, each PDE-cache entry is referenced by a 27-bit value and is used for linear addresses for which bits 47:21 have that value.
  - For 5-level paging, each PDE-cache entry is referenced by a 36-bit value and is used for linear addresses for which bits 56:21 have that value.

  A PDE-cache entry contains information from the PML5E, PML4E, PDPTE, and PDE used to translate the relevant linear addresses (for 32-bit paging and PAE paging, only the PDE applies; for 4-level paging, the PML5E does not apply):
  - The physical address from the PDE (the address of the page table). (No PDE-cache entry is created for a PDE that maps a page.)
  - The logical-AND of the R/W flags in the PML5E, PML4E, PDPTE, and PDE.
  - The logical-AND of the U/S flags in the PML5E, PML4E, PDPTE, and PDE.
  - The logical-OR of the XD flags in the PML5E, PML4E, PDPTE, and PDE.
  - The values of the PCD and PWT flags of the PDE.

  The following items detail how a processor may use the PDE cache (references below to PML5Es, PML4Es, and PDPTEs apply only to 4-level paging and to 5-level paging, as appropriate):
  - If the processor has a PDE-cache entry for a linear address, it may use that entry when translating the linear address (instead of the PML5E, PML4E, PDPTE, and PDE in memory).
  - The processor does not create a PDE-cache entry unless the P flags are 1, the PS flags are 0, and the reserved bits are 0 in the PML5E, PML4E, PDPTE, and PDE in memory.
  - The processor does not create a PDE-cache entry unless the accessed flag is 1 in the PML5E, PML4E, PDPTE, and PDE in memory; before caching a translation, the processor sets any accessed flags that are not already 1.
  - The processor may create a PDE-cache entry even if there are no translations for any linear address that might use that entry.
  - If the processor creates a PDE-cache entry, the processor may retain it unmodified even if software subsequently modifies the corresponding PML5E, PML4E, PDPTE, or PDE in memory.

Information from a paging-structure entry can be included in entries in the paging-structure caches for other paging-structure entries referenced by the original entry. For example, if the R/W flag is 0 in a PML4E, then the R/W flag will be 0 in any PDPTE-cache entry for a PDPTE from the page-directory-pointer table referenced by that PML4E. This is because the R/W flag of each such PDPTE-cache entry is the logical-AND of the R/W flags in the appropriate PML4E and PDPTE.

On processors that support HLAT paging (see Section 5.5.1), each entry in a paging-structure cache indicates whether the entry was cached during ordinary paging or HLAT paging. When the processor commences linear-address translation using ordinary paging (respectively, HLAT paging), it will use only entries that indicate that they were cached during ordinary paging (respectively, HLAT paging).

Entries that were cached during HLAT paging also include the restart flag (bit 11) of the original paging-structure entry. When the processor commences HLAT paging using such an entry, it immediately restarts (using ordinary paging) if this cached restart flag is 1.

The paging-structure caches contain information only from paging-structure entries that reference other paging structures (and not those that map pages). Because the G flag is not used in such paging-structure entries, the global-page feature does not affect the behavior of the paging-structure caches.

The processor may create entries in paging-structure caches for translations required for prefetches and for accesses that are a result of speculative execution that would never actually occur in the executed code path.

As noted in Section 5.10.1, any entries created in paging-structure caches by a logical processor are associated with the current PCID.

A processor may or may not implement any of the paging-structure caches. Software should rely on neither their presence nor their absence. The processor may invalidate entries in these caches at any time. Because the processor may create the cache entries at the time of translation and not update them following subsequent modifications to the paging structures in memory, software should take care to invalidate the cache entries appropriately when causing such modifications. The invalidation of TLBs and the paging-structure caches is described in Section 5.10.4.

5.10.3.2 Using the Paging-Structure Caches to Translate Linear Addresses

When a linear address is accessed, the processor uses a procedure such as the following to determine the physical address to which it translates and whether the access should be allowed:

- If the processor finds a TLB entry that is for the page number of the linear address and that is associated with the current PCID (or which is global), it may use the physical address, access rights, and other attributes from that entry.
- If the processor does not find a relevant TLB entry, it may use the upper bits of the linear address to select an entry from the PDE cache that is associated with the current PCID (Section 5.10.3.1 indicates which bits are used in each paging mode). It can then use that entry to complete the translation process (locating a PTE, etc.) as if it had traversed the PDE (and, for 4-level paging and 5-level paging, the PDPTE, PML4E, and PML5E, as appropriate) corresponding to the PDE-cache entry.
- The following items apply when 4-level paging or 5-level paging is used:
  - If the processor does not find a relevant TLB entry or PDE-cache entry, it may use the upper bits of the linear address (for 4-level paging, bits 47:30; for 5-level paging, bits 56:30) to select an entry from the PDPTE cache that is associated with the current PCID. It can then use that entry to complete the translation process (locating a PDE, etc.) as if it had traversed the PDPTE, the PML4E, and (for 5-level paging) the PML5E corresponding to the PDPTE-cache entry.
  - If the processor does not find a relevant TLB entry, PDE-cache entry, or PDPTE-cache entry, it may use the upper bits of the linear address (for 4-level paging, bits 47:39; for 5-level paging, bits 56:39) to select an entry from the PML4E cache that is associated with the current PCID. It can then use that entry to complete the translation process (locating a PDPTE, etc.) as if it had traversed the corresponding PML4E.
  - With 5-level paging, if the processor does not find a relevant TLB entry, PDE-cache entry, PDPTE-cache entry, or PML4E-cache entry, it may use bits 56:48 of the linear address to select an entry from the PML5E cache that is associated with the current PCID. It can then use that entry to complete the translation process (locating a PML4E, etc.) as if it had traversed the corresponding PML5E.

(Any of the above steps would be skipped if the processor does not support the cache in question.)

If the processor does not find a TLB or paging-structure-cache entry for the linear address, it uses the linear address to traverse the entire paging-structure hierarchy, as described in Section 5.3, Section 5.4.2, and Section 5.5.

5.10.3.3 Multiple Cached Entries for a Single Paging-Structure Entry

The paging-structure caches and TLBs may contain multiple entries associated with a single PCID and with information derived from a single paging-structure entry. The following items give some examples for 4-level paging:

- Suppose that two PML4Es contain the same physical address and thus reference the same page-directory-pointer table. Any PDPTE in that table may result in two PDPTE-cache entries, each associated with a different set of linear addresses. Specifically, suppose that the n_1^{th} and n_2^{th} entries in the PML4 table contain the same physical address. This implies that the physical address in the mth PDPTE in the page-directory-pointer table would appear in the PDPTE-cache entries associated with both p_1 and p_2, where $(p_1 \gg 9) = n_1$, $(p_2 \gg 9) = n_2$, and $(p_1 \& 1FFH) = (p_2 \& 1FFH) = m$. This is because both PDPTE-cache entries use the same PDPTE, one resulting from a reference from the n_1^{th} PML4E and one from the n_2^{th} PML4E.
- Suppose that the first PML4E (i.e., the one in position 0) contains the physical address X in CR3 (the physical address of the PML4 table). This implies the following:

— Any PML4-cache entry associated with linear addresses with 0 in bits 47:39 contains address X.

— Any PDPTE-cache entry associated with linear addresses with 0 in bits 47:30 contains address X. This is because the translation for a linear address for which the value of bits 47:30 is 0 uses the value of bits 47:39 (0) to locate a page-directory-pointer table at address X (the address of the PML4 table). It then uses the value of bits 38:30 (also 0) to find address X again and to store that address in the PDPTE-cache entry.

— Any PDE-cache entry associated with linear addresses with 0 in bits 47:21 contains address X for similar reasons.

— Any TLB entry for page number 0 (associated with linear addresses with 0 in bits 47:12) translates to page frame X » 12 for similar reasons.

The same PML4E contributes its address X to all these cache entries because the self-referencing nature of the entry causes it to be used as a PML4E, a PDPTE, a PDE, and a PTE.

5.10.4 Invalidation of TLBs and Paging-Structure Caches

As noted in Section 5.10.2 and Section 5.10.3, the processor may create entries in the TLBs and the paging-structure caches when linear addresses are translated, and it may retain these entries even after the paging structures used to create them have been modified. To ensure that linear-address translation uses the modified paging structures, software should take action to invalidate any cached entries that may contain information that has since been modified.

5.10.4.1 Operations that Invalidate TLBs and Paging-Structure Caches

The following instructions invalidate entries in the TLBs and the paging-structure caches:

- INVLPG. This instruction takes a single operand, which is a linear address. The instruction invalidates any TLB entries that are for a page number corresponding to the linear address and that are associated with the current PCID. It also invalidates any global TLB entries with that page number, regardless of PCID (see Section 5.10.2.4).[1] INVLPG also invalidates all entries in all paging-structure caches associated with the current PCID, regardless of the linear addresses to which they correspond.
- INVPCID. The operation of this instruction is based on instruction operands, called the INVPCID type and the INVPCID descriptor. Four INVPCID types are currently defined:
  — Individual-address. If the INVPCID type is 0, the logical processor invalidates mappings—except global translations—associated with the PCID specified in the INVPCID descriptor and that would be used to translate the linear address specified in the INVPCID descriptor.[2] (The instruction may also invalidate global translations, as well as mappings associated with other PCIDs and for other linear addresses.)
  — Single-context. If the INVPCID type is 1, the logical processor invalidates all mappings—except global translations—associated with the PCID specified in the INVPCID descriptor. (The instruction may also invalidate global translations, as well as mappings associated with other PCIDs.)
  — All-context, including globals. If the INVPCID type is 2, the logical processor invalidates mappings—including global translations—associated with all PCIDs.
  — All-context. If the INVPCID type is 3, the logical processor invalidates mappings—except global translations—associated with all PCIDs. (The instruction may also invalidate global translations.)

  See Chapter 3 of the *Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 2A* for details of the INVPCID instruction.
- MOV to CR0. The instruction invalidates all TLB entries (including global entries) and all entries in all paging-structure caches (for all PCIDs) if it changes the value of CR0.PG from 1 to 0.
- MOV to CR3. The behavior of the instruction depends on the value of CR4.PCIDE:

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 5.10.2.3), the instruction invalidates all of them.
2. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 5.10.2.3), the instruction invalidates all of them.

  - If CR4.PCIDE = 0, the instruction invalidates all TLB entries associated with PCID 000H except those for global pages. It also invalidates all entries in all paging-structure caches associated with PCID 000H.
  - If CR4.PCIDE = 1 and bit 63 of the instruction's source operand is 0, the instruction invalidates all TLB entries associated with the PCID specified in bits 11:0 of the instruction's source operand except those for global pages. It also invalidates all entries in all paging-structure caches associated with that PCID. It is not required to invalidate entries in the TLBs and paging-structure caches that are associated with other PCIDs.
  - If CR4.PCIDE = 1 and bit 63 of the instruction's source operand is 1, the instruction is not required to invalidate any TLB entries or entries in paging-structure caches.
- MOV to CR4. The behavior of the instruction depends on the bits being modified:
  - The instruction invalidates all TLB entries (including global entries) and all entries in all paging-structure caches (for all PCIDs) if (1) it changes the value of CR4.PGE;[1] or (2) it changes the value of the CR4.PCIDE from 1 to 0.
  - The instruction invalidates all TLB entries and all entries in all paging-structure caches for the current PCID if (1) it changes the value of CR4.PAE; or (2) it changes the value of CR4.SMEP from 0 to 1.
- Task switch. If a task switch changes the value of CR3, it invalidates all TLB entries associated with PCID 000H except those for global pages. It also invalidates all entries in all paging-structure caches associated with PCID 000H.[2]
- VMX transitions. See Section 5.11.1.

The processor is always free to invalidate additional entries in the TLBs and paging-structure caches. The following are some examples:

- INVLPG may invalidate TLB entries for pages other than the one corresponding to its linear-address operand. It may invalidate TLB entries and paging-structure-cache entries associated with PCIDs other than the current PCID.
- INVPCID may invalidate TLB entries for pages other than the one corresponding to the specified linear address. It may invalidate TLB entries and paging-structure-cache entries associated with PCIDs other than the specified PCID.
- MOV to CR0 may invalidate TLB entries even if CR0.PG is not changing. For example, this may occur if either CR0.CD or CR0.NW is modified.
- MOV to CR3 may invalidate TLB entries for global pages. If CR4.PCIDE = 1 and bit 63 of the instruction's source operand is 0, it may invalidate TLB entries and entries in the paging-structure caches associated with PCIDs other than the PCID it is establishing. It may invalidate entries if CR4.PCIDE = 1 and bit 63 of the instruction's source operand is 1.
- MOV to CR4 may invalidate TLB entries when changing CR4.PSE or when changing CR4.SMEP from 1 to 0.
- On a processor supporting Hyper-Threading Technology, invalidations performed on one logical processor may invalidate entries in the TLBs and paging-structure caches used by other logical processors.

(Other instructions and operations may invalidate entries in the TLBs and the paging-structure caches, but the instructions identified above are recommended.)

In addition to the instructions identified above, page faults invalidate entries in the TLBs and paging-structure caches. In particular, a page-fault exception resulting from an attempt to use a linear address will invalidate any TLB entries that are for a page number corresponding to that linear address and that are associated with the current PCID. It also invalidates all entries in the paging-structure caches that would be used for that linear address and that are associated with the current PCID.[3] These invalidations ensure that the page-fault exception will not recur (if the faulting instruction is re-executed) if it would not be caused by the contents of the paging structures

1. If CR4.PGE is changing from 0 to 1, there were no global TLB entries before the execution; if CR4.PGE is changing from 1 to 0, there will be no global TLB entries after the execution.
2. Task switches do not occur in IA-32e mode and thus cannot occur with 4-level paging. Since CR4.PCIDE can be set only with 4-level paging, task switches occur only with CR4.PCIDE = 0.
3. Unlike INVLPG, page faults need not invalidate **all** entries in the paging-structure caches, only those that would be used to translate the faulting linear address.

in memory (and if, therefore, it resulted from cached entries that were not invalidated after the paging structures were modified in memory).

As noted in Section 5.10.2, some processors may choose to cache multiple smaller-page TLB entries for a translation specified by the paging structures to use a page larger than 4 KBytes. There is no way for software to be aware that multiple translations for smaller pages have been used for a large page. The INVLPG instruction and page faults provide the same assurances that they provide when a single TLB entry is used: they invalidate all TLB entries corresponding to the translation specified by the paging structures.

5.10.4.2 Recommended Invalidation

The following items provide some recommendations regarding when software should perform invalidations:

- If software modifies a paging-structure entry that maps a page (rather than referencing another paging structure), it should execute INVLPG for any linear address with a page number whose translation uses that paging-structure entry.[1]

  (If the paging-structure entry may be used in the translation of different page numbers — see Section 5.10.3.3 — software should execute INVLPG for linear addresses with each of those page numbers; alternatively, it could use MOV to CR3 or MOV to CR4.)
- If software modifies a paging-structure entry that references another paging structure, it may use one of the following approaches depending upon the types and number of translations controlled by the modified entry:
  - Execute INVLPG for linear addresses with each of the page numbers with translations that would use the entry. However, if no page numbers that would use the entry have translations (e.g., because the P flags are 0 in all entries in the paging structure referenced by the modified entry), it remains necessary to execute INVLPG at least once.
  - Execute MOV to CR3 if the modified entry controls no global pages.
  - Execute MOV to CR4 to modify CR4.PGE.
- If CR4.PCIDE = 1 and software modifies a paging-structure entry that does not map a page or in which the G flag (bit 8) is 0, additional steps are required if the entry may be used for PCIDs other than the current one. Any one of the following suffices:
  - Execute MOV to CR4 to modify CR4.PGE, either immediately or before again using any of the affected PCIDs. For example, software could use different (previously unused) PCIDs for the processes that used the affected PCIDs.
  - For each affected PCID, execute MOV to CR3 to make that PCID current (and to load the address of the appropriate PML4 table). If the modified entry controls no global pages and bit 63 of the source operand to MOV to CR3 was 0, no further steps are required. Otherwise, execute INVLPG for linear addresses with each of the page numbers with translations that would use the entry; if no page numbers that would use the entry have translations, execute INVLPG at least once.
- If software using PAE paging modifies a PDPTE, it should reload CR3 with the register's current value to ensure that the modified PDPTE is loaded into the corresponding PDPTE register (see Section 5.4.1).
- If the nature of the paging structures is such that a single entry may be used for multiple purposes (see Section 5.10.3.3), software should perform invalidations for all of these purposes. For example, if a single entry might serve as both a PDE and PTE, it may be necessary to execute INVLPG with two (or more) linear addresses, one that uses the entry as a PDE and one that uses it as a PTE. (Alternatively, software could use MOV to CR3 or MOV to CR4.)
- As noted in Section 5.10.2, the TLBs may subsequently contain multiple translations for the address range if software modifies the paging structures so that the page size used for a 4-KByte range of linear addresses changes. A reference to a linear address in the address range may use any of these translations.

  Software wishing to prevent this uncertainty should not write to a paging-structure entry in a way that would change, for any linear address, both the page size and either the page frame, access rights, or other attributes. It can instead use the following algorithm: first clear the P flag in the relevant paging-structure entry (e.g.,

1. One execution of INVLPG is sufficient even for a page with size greater than 4 KBytes.

PDE); then invalidate any translations for the affected linear addresses (see above); and then modify the relevant paging-structure entry to set the P flag and establish modified translation(s) for the new page size.

- Software should clear bit 63 of the source operand to a MOV to CR3 instruction that establishes a PCID that had been used earlier for a different linear-address space (e.g., with a different value in bits 51:12 of CR3). This ensures invalidation of any information that may have been cached for the previous linear-address space.

  This assumes that both linear-address spaces use the same global pages and that it is thus not necessary to invalidate any global TLB entries. If that is not the case, software should invalidate those entries by executing MOV to CR4 to modify CR4.PGE.

5.10.4.3 Optional Invalidation

The following items describe cases in which software may choose not to invalidate and the potential consequences of that choice:

- If a paging-structure entry is modified to change the P flag from 0 to 1, no invalidation is necessary. This is because no TLB entry or paging-structure cache entry is created with information from a paging-structure entry in which the P flag is 0.[1]
- If a paging-structure entry is modified to change the accessed flag from 0 to 1, no invalidation is necessary (assuming that an invalidation was performed the last time the accessed flag was changed from 1 to 0). This is because no TLB entry or paging-structure cache entry is created with information from a paging-structure entry in which the accessed flag is 0.
- If a paging-structure entry is modified to change the R/W flag from 0 to 1, failure to perform an invalidation may result in a "spurious" page-fault exception (e.g., in response to an attempted write access) but no other adverse behavior. Such an exception will occur at most once for each affected linear address (see Section 5.10.4.1).
- If CR4.SMEP = 0 and a paging-structure entry is modified to change the U/S flag from 0 to 1, failure to perform an invalidation may result in a "spurious" page-fault exception (e.g., in response to an attempted user-mode access) but no other adverse behavior. Such an exception will occur at most once for each affected linear address (see Section 5.10.4.1).
- If a paging-structure entry is modified to change the XD flag from 1 to 0, failure to perform an invalidation may result in a "spurious" page-fault exception (e.g., in response to an attempted instruction fetch) but no other adverse behavior. Such an exception will occur at most once for each affected linear address (see Section 5.10.4.1).
- If a paging-structure entry is modified to change the accessed flag from 1 to 0, failure to perform an invalidation may result in the processor not setting that bit in response to a subsequent access to a linear address whose translation uses the entry. Software cannot interpret the bit being clear as an indication that such an access has not occurred.
- If software modifies a paging-structure entry that identifies the final physical address for a linear address (either a PTE or a paging-structure entry in which the PS flag is 1) to change the dirty flag from 1 to 0, failure to perform an invalidation may result in the processor not setting that bit in response to a subsequent write to a linear address whose translation uses the entry. Software cannot interpret the bit being clear as an indication that such a write has not occurred.
- The read of a paging-structure entry in translating an address being used to fetch an instruction may appear to execute before an earlier write to that paging-structure entry if there is no serializing instruction between the write and the instruction fetch. Note that the invalidating instructions identified in Section 5.10.4.1 are all serializing instructions.
- Section 5.10.3.3 describes situations in which a single paging-structure entry may contain information cached in multiple entries in the paging-structure caches. Because all entries in these caches are invalidated by any execution of INVLPG, it is not necessary to follow the modification of such a paging-structure entry by executing INVLPG multiple times solely for the purpose of invalidating these multiple cached entries. (It may be necessary to do so to invalidate multiple TLB entries.)

1. If it is also the case that no invalidation was performed the last time the P flag was changed from 1 to 0, the processor may use a TLB entry or paging-structure cache entry that was created when the P flag had earlier been 1.

5.10.4.4 Delayed Invalidation

Required invalidations may be delayed under some circumstances. Software developers should understand that, between the modification of a paging-structure entry and execution of the invalidation instruction recommended in Section 5.10.4.2, the processor may use translations based on either the old value or the new value of the paging-structure entry. The following items describe some of the potential consequences of delayed invalidation:

- If a paging-structure entry is modified to change the P flag from 1 to 0, an access to a linear address whose translation is controlled by this entry may or may not cause a page-fault exception.
- If a paging-structure entry is modified to change the R/W flag from 0 to 1, write accesses to linear addresses whose translation is controlled by this entry may or may not cause a page-fault exception.
- If a paging-structure entry is modified to change the U/S flag from 0 to 1, user-mode accesses to linear addresses whose translation is controlled by this entry may or may not cause a page-fault exception.
- If a paging-structure entry is modified to change the XD flag from 1 to 0, instruction fetches from linear addresses whose translation is controlled by this entry may or may not cause a page-fault exception.

As noted in Section 10.1.1, an x87 instruction or an SSE instruction that accesses data larger than a quadword may be implemented using multiple memory accesses. If such an instruction stores to memory and invalidation has been delayed, some of the accesses may complete (writing to memory) while another causes a page-fault exception.[1] In this case, the effects of the completed accesses may be visible to software even though the overall instruction caused a fault.

In some cases, the consequences of delayed invalidation may not affect software adversely. For example, when freeing a portion of the linear-address space (by marking paging-structure entries "not present"), invalidation using INVLPG may be delayed if software does not re-allocate that portion of the linear-address space or the memory that had been associated with it. However, because of speculative execution (or errant software), there may be accesses to the freed portion of the linear-address space before the invalidations occur. In this case, the following can happen:

- Reads can occur to the freed portion of the linear-address space. Therefore, invalidation should not be delayed for an address range that has read side effects.
- The processor may retain entries in the TLBs and paging-structure caches for an extended period of time. Software should not assume that the processor will not use entries associated with a linear address simply because time has passed.
- As noted in Section 5.10.3.1, the processor may create an entry in a paging-structure cache even if there are no translations for any linear address that might use that entry. Thus, if software has marked "not present" all entries in a page table, the processor may subsequently create a PDE-cache entry for the PDE that references that page table (assuming that the PDE itself is marked "present").
- If software attempts to write to the freed portion of the linear-address space, the processor might not generate a page fault. (Such an attempt would likely be the result of a software error.) For that reason, the page frames previously associated with the freed portion of the linear-address space should not be reallocated for another purpose until the appropriate invalidations have been performed.

5.10.5 Propagation of Paging-Structure Changes to Multiple Processors

As noted in Section 5.10.4, software that modifies a paging-structure entry may need to invalidate entries in the TLBs and paging-structure caches that were derived from the modified entry before it was modified. In a system containing more than one logical processor, software must account for the fact that there may be entries in the TLBs and paging-structure caches of logical processors other than the one used to modify the paging-structure entry. The process of propagating the changes to a paging-structure entry is commonly referred to as "TLB shootdown."

TLB shootdown can be done using memory-based semaphores and/or interprocessor interrupts (IPI). The following items describe a simple but inefficient example of a TLB shootdown algorithm for processors supporting the Intel-64 and IA-32 architectures:

1. If the accesses are to different pages, this may occur even if invalidation has not been delayed.

1. Begin barrier: Stop all but one logical processor; that is, cause all but one to execute the HLT instruction or to enter a spin loop.
2. Allow the active logical processor to change the necessary paging-structure entries.
3. Allow all logical processors to perform invalidations appropriate to the modifications to the paging-structure entries.
4. Allow all logical processors to resume normal operation.

Alternative, performance-optimized, TLB shootdown algorithms may be developed; however, software developers must take care to ensure that the following conditions are met:

- All logical processors that are using the paging structures that are being modified must participate and perform appropriate invalidations after the modifications are made.
- If the modifications to the paging-structure entries are made before the barrier or if there is no barrier, the operating system must ensure one of the following: (1) that the affected linear-address range is not used between the time of modification and the time of invalidation; or (2) that it is prepared to deal with the consequences of the affected linear-address range being used during that period. For example, if the operating system does not allow pages being freed to be reallocated for another purpose until after the required invalidations, writes to those pages by errant software will not unexpectedly modify memory that is in use.
- Software must be prepared to deal with reads, instruction fetches, and prefetch requests to the affected linear-address range that are a result of speculative execution that would never actually occur in the executed code path.

When multiple logical processors are using the same linear-address space at the same time, they must coordinate before any request to modify the paging-structure entries that control that linear-address space. In these cases, the barrier in the TLB shootdown routine may not be required. For example, when freeing a range of linear addresses, some other mechanism can assure no logical processor is using that range before the request to free it is made. In this case, a logical processor freeing the range can clear the P flags in the PTEs associated with the range, free the physical page frames associated with the range, and then signal the other logical processors using that linear-address space to perform the necessary invalidations. All the affected logical processors must complete their invalidations before the linear-address range and the physical page frames previously associated with that range can be reallocated.

5.11 INTERACTIONS WITH VIRTUAL-MACHINE EXTENSIONS (VMX)

The architecture for virtual-machine extensions (VMX) includes features that interact with paging. Section 5.11.1 discusses ways in which VMX-specific control transfers, called VMX transitions specially affect paging. Section 5.11.2 gives an overview of VMX features specifically designed to support address translation.

5.11.1 VMX Transitions

The VMX architecture defines two control transfers called **VM entries** and **VM exits**; collectively, these are called **VMX transitions**. VM entries and VM exits are described in detail in Chapter 27 and Chapter 28, respectively, in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C. The following items identify paging-related details:

- VMX transitions modify the CR0 and CR4 registers and the IA32_EFER MSR concurrently. For this reason, they allow transitions between paging modes that would not otherwise be possible:
  - VM entries allow transitions from 4-level paging directly to either 32-bit paging or PAE paging.
  - VM exits allow transitions from either 32-bit paging or PAE paging directly to 4-level paging or 5-level paging.
- VMX transitions that result in PAE paging load the PDPTE registers (see Section 5.4.1) as follows:
  - VM entries load the PDPTE registers either from the physical address being loaded into CR3 or from the virtual-machine control structure (VMCS); see Section 28.3.2.4.
  - VM exits load the PDPTE registers from the physical address being loaded into CR3; see Section 29.5.4.

- VMX transitions invalidate the TLBs and paging-structure caches based on certain control settings. See Section 28.3.2.5 and Section 29.5.5 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C.

5.11.2 VMX Support for Address Translation

Chapter 30, "VMX Support for Address Translation," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C, describes two features of the virtual-machine extensions (VMX) that interact directly with paging. These are **virtual-processor identifiers** (**VPIDs**) and the **extended page table** mechanism (**EPT**).

VPIDs provide a way for software to identify to the processor the address spaces for different "virtual processors." The processor may use this identification to maintain concurrently information for multiple address spaces in its TLBs and paging-structure caches, even when non-zero PCIDs are not being used. See Section 30.1 for details.

When EPT is in use, the addresses in the paging-structures are not used as physical addresses to access memory and memory-mapped I/O. Instead, they are treated as **guest-physical** addresses and are translated through a set of EPT paging structures to produce physical addresses. EPT can also specify its own access rights and memory typing; these are used on conjunction with those specified in this chapter. See Section 30.3 for more information.

Both VPIDs and EPT may change the way that a processor maintains information in TLBs and paging structure caches and the ways in which software can manage that information. Some of the behaviors documented in Section 5.10 may change. See Section 30.4 for details.

5.12 USING PAGING FOR VIRTUAL MEMORY

With paging, portions of the linear-address space need not be mapped to the physical-address space; data for the unmapped addresses can be stored externally (e.g., on disk). This method of mapping the linear-address space is referred to as virtual memory or demand-paged virtual memory.

Paging divides the linear address space into fixed-size pages that can be mapped into the physical-address space and/or external storage. When a program (or task) references a linear address, the processor uses paging to translate the linear address into a corresponding physical address if such an address is defined.

If the page containing the linear address is not currently mapped into the physical-address space, the processor generates a page-fault exception as described in Section 5.7. The handler for page-fault exceptions typically directs the operating system or executive to load data for the unmapped page from external storage into physical memory (perhaps writing a different page from physical memory out to external storage in the process) and to map it using paging (by updating the paging structures). When the page has been loaded into physical memory, a return from the exception handler causes the instruction that generated the exception to be restarted.

Paging differs from segmentation through its use of fixed-size pages. Unlike segments, which usually are the same size as the code or data structures they hold, pages have a fixed size. If segmentation is the only form of address translation used, a data structure present in physical memory will have all of its parts in memory. If paging is used, a data structure can be partly in memory and partly in disk storage.

5.13 MAPPING SEGMENTS TO PAGES

The segmentation and paging mechanisms provide support for a wide variety of approaches to memory management. When segmentation and paging are combined, segments can be mapped to pages in several ways. To implement a flat (unsegmented) addressing environment, for example, all the code, data, and stack modules can be mapped to one or more large segments (up to 4-GBytes) that share same range of linear addresses (see Figure 3-2 in Section 3.2.2). Here, segments are essentially invisible to applications and the operating-system or executive. If paging is used, the paging mechanism can map a single linear-address space (contained in a single segment) into virtual memory. Alternatively, each program (or task) can have its own large linear-address space (contained in its own segment), which is mapped into virtual memory through its own paging structures.

Segments can be smaller than the size of a page. If one of these segments is placed in a page which is not shared with another segment, the extra memory is wasted. For example, a small data structure, such as a 1-Byte semaphore, occupies 4 KBytes if it is placed in a page by itself. If many semaphores are used, it is more efficient to pack them into a single page.

The Intel-64 and IA-32 architectures do not enforce correspondence between the boundaries of pages and segments. A page can contain the end of one segment and the beginning of another. Similarly, a segment can contain the end of one page and the beginning of another.

Memory-management software may be simpler and more efficient if it enforces some alignment between page and segment boundaries. For example, if a segment which can fit in one page is placed in two pages, there may be twice as much paging overhead to support access to that segment.

One approach to combining paging and segmentation that simplifies memory-management software is to give each segment its own page table, as shown in Figure 5-13. This convention gives the segment a single entry in the page directory, and this entry provides the access control information for paging the entire segment.

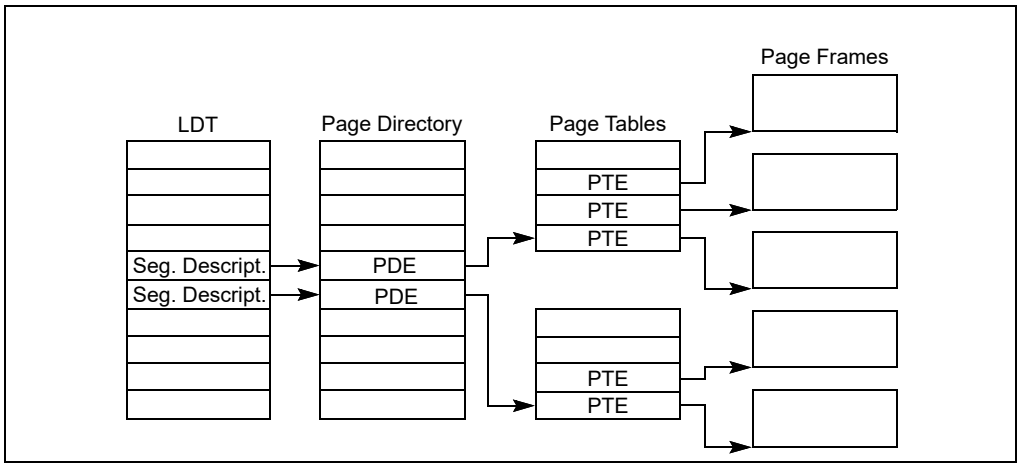


Figure 5-13. Memory Management Convention That Assigns a Page Table to Each Segment

CHAPTER 6
PROTECTION

In protected mode, the Intel 64 and IA-32 architectures provide a protection mechanism that operates at both the segment level and the page level. This protection mechanism provides the ability to limit access to certain segments or pages based on privilege levels (four privilege levels for segments and two privilege levels for pages). For example, critical operating-system code and data can be protected by placing them in more privileged segments than those that contain applications code. The processor's protection mechanism will then prevent application code from accessing the operating-system code and data in any but a controlled, defined manner.

Segment and page protection can be used at all stages of software development to assist in localizing and detecting design problems and bugs. It can also be incorporated into end-products to offer added robustness to operating systems, utilities software, and applications software.

When the protection mechanism is used, each memory reference is checked to verify that it satisfies various protection checks. All checks are made before the memory cycle is started; any violation results in an exception. Because checks are performed in parallel with address translation, there is no performance penalty. The protection checks that are performed fall into the following categories:

- Limit checks.
- Type checks.
- Privilege level checks.
- Restriction of addressable domain.
- Restriction of procedure entry-points.
- Restriction of instruction set.

All protection violation results in an exception being generated. See Chapter 7, "Interrupt and Exception Handling," for an explanation of the exception mechanism. This chapter describes the protection mechanism and the violations which lead to exceptions.

The following sections describe the protection mechanism available in protected mode. See Chapter 22, "8086 Emulation," for information on protection in real-address and virtual-8086 mode.

6.1 ENABLING AND DISABLING SEGMENT AND PAGE PROTECTION

Setting the PE flag in register CR0 causes the processor to switch to protected mode, which in turn enables the segment-protection mechanism. Once in protected mode, there is no control bit for turning the protection mechanism on or off. The part of the segment-protection mechanism that is based on privilege levels can essentially be disabled while still in protected mode by assigning a privilege level of 0 (most privileged) to all segment selectors and segment descriptors. This action disables the privilege level protection barriers between segments, but other protection checks such as limit checking and type checking are still carried out.

Page-level protection is automatically enabled when paging is enabled (by setting the PG flag in register CR0). Here again there is no mode bit for turning off page-level protection once paging is enabled. However, page-level protection can be disabled by performing the following operations:

- Clear the WP flag in control register CR0.
- Set the read/write (R/W) and user/supervisor (U/S) flags for each page-directory and page-table entry.

This action makes each page a writable, user page, which in effect disables page-level protection.

6.2 FIELDS AND FLAGS USED FOR SEGMENT-LEVEL AND PAGE-LEVEL PROTECTION

The processor's protection mechanism uses the following fields and flags in the system data structures to control access to segments and pages:

- **Descriptor type (S) flag —** (Bit 12 in the second doubleword of a segment descriptor.) Determines if the segment descriptor is for a system segment or a code or data segment.
- **Type field** — (Bits 8 through 11 in the second doubleword of a segment descriptor.) Determines the type of code, data, or system segment.
- **Limit field** — (Bits 0 through 15 of the first doubleword and bits 16 through 19 of the second doubleword of a segment descriptor.) Determines the size of the segment, along with the G flag and E flag (for data segments).
- **G flag —** (Bit 23 in the second doubleword of a segment descriptor.) Determines the size of the segment, along with the limit field and E flag (for data segments).
- **E flag —** (Bit 10 in the second doubleword of a data-segment descriptor.) Determines the size of the segment, along with the limit field and G flag.
- **Descriptor privilege level (DPL) field** — (Bits 13 and 14 in the second doubleword of a segment descriptor.) Determines the privilege level of the segment.
- **Requested privilege level (RPL) field** — (Bits 0 and 1 of any segment selector.) Specifies the requested privilege level of a segment selector.
- **Current privilege level (CPL) field** — (Bits 0 and 1 of the CS segment register.) Indicates the privilege level of the currently executing program or procedure. The term current privilege level (CPL) refers to the setting of this field.
- **User/supervisor (U/S) flag** — (Bit 2 of paging-structure entries.) Determines the type of page: user or supervisor.
- **Read/write (R/W) flag** — (Bit 1 of paging-structure entries.) Determines the type of access allowed to a page: read-only or read/write.
- **Execute-disable (XD) flag** — (Bit 63 of certain paging-structure entries.) Determines the type of access allowed to a page: executable or not-executable.

Figure 6-1 shows the location of the various fields and flags in the data-, code-, and system-segment descriptors; Figure 3-6 shows the location of the RPL (or CPL) field in a segment selector (or the CS register); and Chapter 5 identifies the locations of the U/S, R/W, and XD flags in the paging-structure entries.

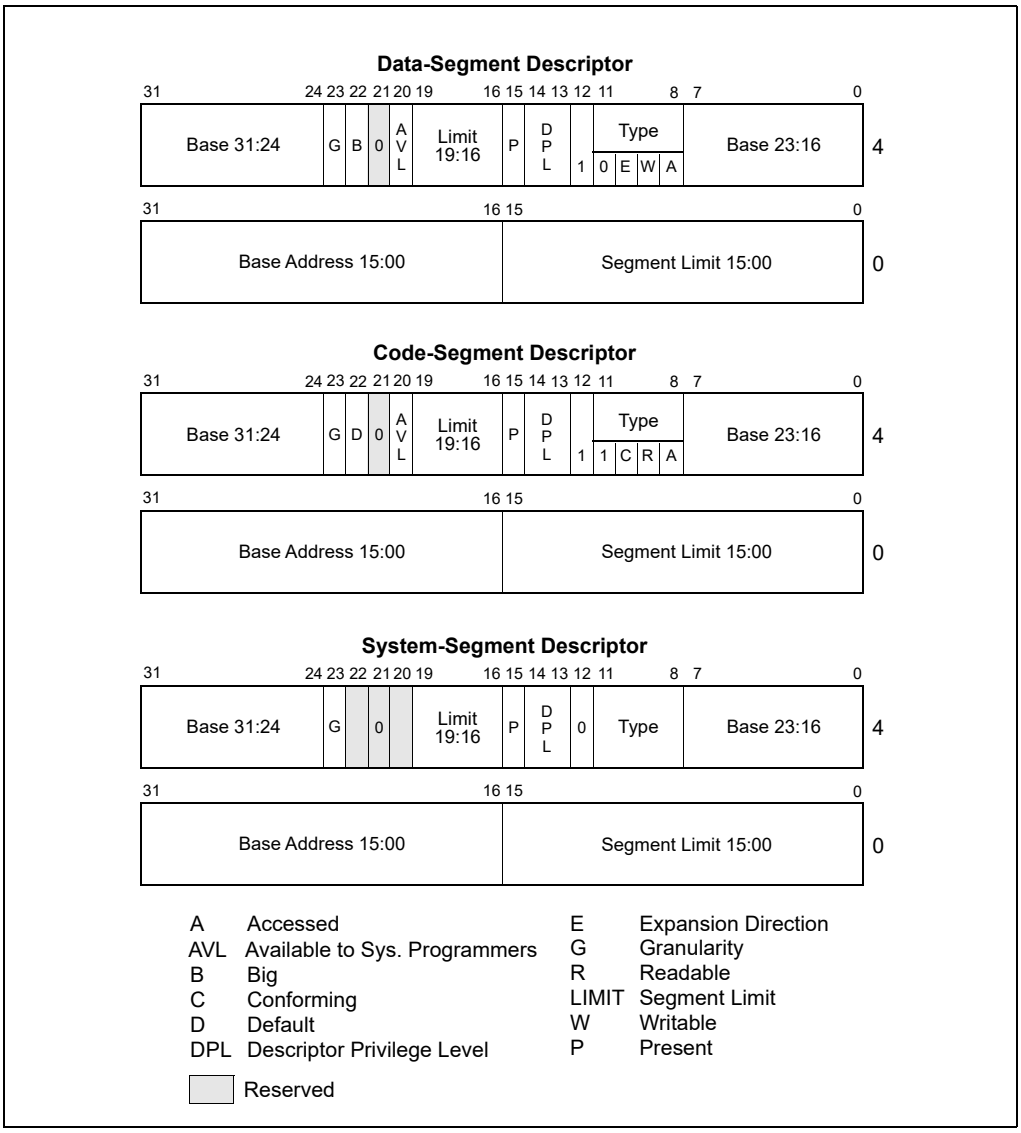


Figure 6-1. Descriptor Fields Used for Protection

Many different styles of protection schemes can be implemented with these fields and flags. When the operating system creates a descriptor, it places values in these fields and flags in keeping with the particular protection style chosen for an operating system or executive. Application programs do not generally access or modify these fields and flags.

The following sections describe how the processor uses these fields and flags to perform the various categories of checks described in the introduction to this chapter.

6.2.1 Code-Segment Descriptor in 64-bit Mode

Code segments continue to exist in 64-bit mode even though, for address calculations, the segment base is treated as zero. Some code-segment (CS) descriptor content (the base address and limit fields) is ignored; the remaining fields function normally (except for the readable bit in the type field).

Code segment descriptors and selectors are needed in IA-32e mode to establish the processor's operating mode and execution privilege-level. The usage is as follows:

- IA-32e mode uses a previously unused bit in the CS descriptor. Bit 53 is defined as the 64-bit (L) flag and is used to select between 64-bit mode and compatibility mode when IA-32e mode is active (IA32_EFER.LMA = 1). See Figure 6-2.
  - — If CS.L = 0 and IA-32e mode is active, the processor is running in compatibility mode. In this case, CS.D selects the default size for data and addresses. If CS.D = 0, the default data and address size is 16 bits. If CS.D = 1, the default data and address size is 32 bits.
  - — If CS.L = 1 and IA-32e mode is active, the only valid setting is CS.D = 0. This setting indicates a default operand size of 32 bits and a default address size of 64 bits. The CS.L = 1 and CS.D = 1 bit combination is reserved for future use and a #GP fault will be generated on an attempt to use a code segment with these bits set in IA-32e mode.
- In IA-32e mode, the CS descriptor's DPL is used for execution privilege checks (as in legacy 32-bit mode).

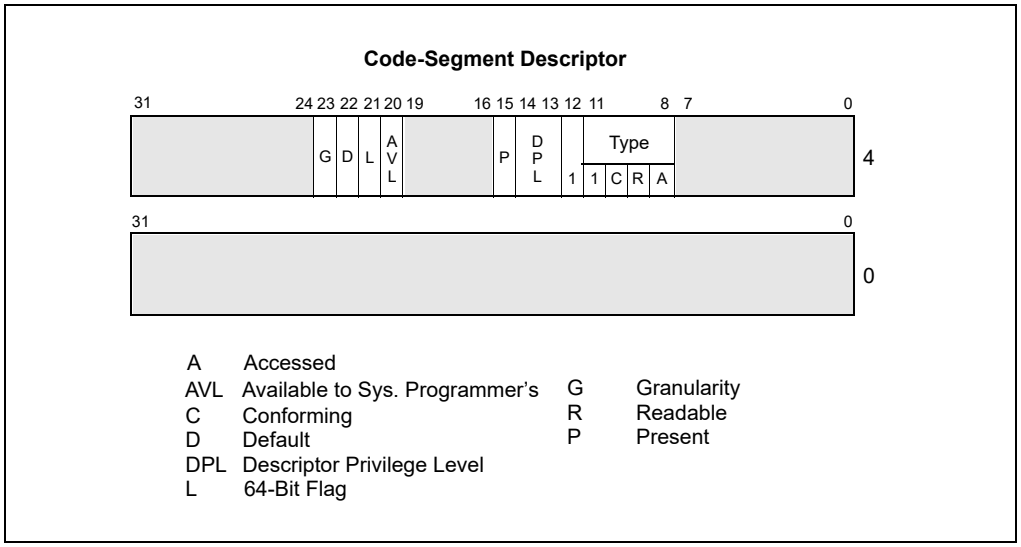


Figure 6-2. Descriptor Fields with Flags used in IA-32e Mode

6.3 LIMIT CHECKING

The limit field of a segment descriptor prevents programs or procedures from addressing memory locations outside the segment. The effective value of the limit depends on the setting of the G (granularity) flag (see Figure 6-1). For data segments, the limit also depends on the E (expansion direction) flag and the B (default stack pointer size and/or upper bound) flag. The E flag is one of the bits in the type field when the segment descriptor is for a data-segment type.

When the G flag is clear (byte granularity), the effective limit is the value of the 20-bit limit field in the segment descriptor. Here, the limit ranges from 0 to FFFFFH (1 MByte). When the G flag is set (4-KByte page granularity), the processor scales the value in the limit field by a factor of 2^{12} (4 KBytes). In this case, the effective limit ranges from FFFH (4 KBytes) to FFFFFFFFH (4 GBytes). Note that when scaling is used (G flag is set), the lower 12 bits of a segment offset (address) are not checked against the limit; for example, note that if the segment limit is 0, offsets 0 through FFFH are still valid.

For all types of segments except expand-down data segments, the effective limit is the last address that is allowed to be accessed in the segment, which is one less than the size, in bytes, of the segment. The processor causes a general-protection exception (or, if the segment is SS, a stack-fault exception) any time an attempt is made to access the following addresses in a segment:

- A byte at an offset greater than the effective limit
- A word at an offset greater than the (effective-limit – 1)
- A doubleword at an offset greater than the (effective-limit – 3)
- A quadword at an offset greater than the (effective-limit – 7)

- A double quadword at an offset greater than the (effective limit – 15)

When the effective limit is FFFFFFFFH (4 GBytes), these accesses may or may not cause the indicated exceptions. Behavior is implementation-specific and may vary from one execution to another.

For expand-down data segments, the segment limit has the same function but is interpreted differently. Here, the effective limit specifies the last address that is not allowed to be accessed within the segment; the range of valid offsets is from (effective-limit + 1) to FFFFFFFFH if the B flag is set and from (effective-limit + 1) to FFFFH if the B flag is clear. An expand-down segment has maximum size when the segment limit is 0.

Limit checking catches programming errors such as runaway code, runaway subscripts, and invalid pointer calculations. These errors are detected when they occur, so identification of the cause is easier. Without limit checking, these errors could overwrite code or data in another segment.

In addition to checking segment limits, the processor also checks descriptor table limits. The GDTR and IDTR registers contain 16-bit limit values that the processor uses to prevent programs from selecting a segment descriptors outside the respective descriptor tables. The LDTR and task registers contain 32-bit segment limit value (read from the segment descriptors for the current LDT and TSS, respectively). The processor uses these segment limits to prevent accesses beyond the bounds of the current LDT and TSS. See Section 3.5.1, "Segment Descriptor Tables," for more information on the GDT and LDT limit fields; see Section 7.10, "Interrupt Descriptor Table (IDT)," for more information on the IDT limit field; and see Section 9.2.4, "Task Register," for more information on the TSS segment limit field.

6.3.1 Limit Checking in 64-bit Mode

In 64-bit mode, the processor does not perform runtime limit checking on code or data segments. However, the processor does check descriptor-table limits.

6.4 TYPE CHECKING

Segment descriptors contain type information in two places:

- The S (descriptor type) flag.
- The type field.

The processor uses this information to detect programming errors that result in an attempt to use a segment or gate in an incorrect or unintended manner.

The S flag indicates whether a descriptor is a system type or a code or data type. The type field provides 4 additional bits for use in defining various types of code, data, and system descriptors. Table 3-1 shows the encoding of the type field for code and data descriptors; Table 3-2 shows the encoding of the field for system descriptors.

The processor examines type information at various times while operating on segment selectors and segment descriptors. The following list gives examples of typical operations where type checking is performed (this list is not exhaustive):

- **When a segment selector is loaded into a segment register** — Certain segment registers can contain only certain descriptor types, for example:
  - The CS register only can be loaded with a selector for a code segment.
  - Segment selectors for code segments that are not readable or for system segments cannot be loaded into data-segment registers (DS, ES, FS, and GS).
  - Only segment selectors of writable data segments can be loaded into the SS register.
- When a segment selector is loaded into the LDTR or task register — For example:
  - The LDTR can only be loaded with a selector for an LDT.
  - The task register can only be loaded with a segment selector for a TSS.
- **When instructions access segments whose descriptors are already loaded into segment registers** — Certain segments can be used by instructions only in certain predefined ways, for example:
  - No instruction may write into an executable segment.

  - No instruction may write into a data segment if it is not writable.
  - No instruction may read an executable segment unless the readable flag is set.
- **When an instruction operand contains a segment selector** — Certain instructions can access segments or gates of only a particular type, for example:
  - A far CALL or far JMP instruction can only access a segment descriptor for a conforming code segment, nonconforming code segment, call gate, task gate, or TSS.
  - The LLDT instruction must reference a segment descriptor for an LDT.
  - The LTR instruction must reference a segment descriptor for a TSS.
  - The LAR instruction must reference a segment or gate descriptor for an LDT, TSS, call gate, task gate, code segment, or data segment.
  - The LSL instruction must reference a segment descriptor for a LDT, TSS, code segment, or data segment.
  - IDT entries must be interrupt, trap, or task gates.
- **During certain internal operations** — For example:
  - On a far call or far jump (executed with a far CALL or far JMP instruction), the processor determines the type of control transfer to be carried out (call or jump to another code segment, a call or jump through a gate, or a task switch) by checking the type field in the segment (or gate) descriptor pointed to by the segment (or gate) selector given as an operand in the CALL or JMP instruction. If the descriptor type is for a code segment or call gate, a call or jump to another code segment is indicated; if the descriptor type is for a TSS or task gate, a task switch is indicated.
  - On a call or jump through a call gate (or on an interrupt- or exception-handler call through a trap or interrupt gate), the processor automatically checks that the segment descriptor being pointed to by the gate is for a code segment.
  - On a call or jump to a new task through a task gate (or on an interrupt- or exception-handler call to a new task through a task gate), the processor automatically checks that the segment descriptor being pointed to by the task gate is for a TSS.
  - On a call or jump to a new task by a direct reference to a TSS, the processor automatically checks that the segment descriptor being pointed to by the CALL or JMP instruction is for a TSS.
  - On return from a nested task (initiated by an IRET instruction), the processor checks that the previous task link field in the current TSS points to a TSS.

6.4.1 Null Segment Selector Checking

Attempting to load a null segment selector (see Section 3.4.2, "Segment Selectors") into the CS or SS segment register generates a general-protection exception (#GP). A null segment selector can be loaded into the DS, ES, FS, or GS register, but any attempt to access a segment through one of these registers when it is loaded with a null segment selector results in a #GP exception being generated. Loading unused data-segment registers with a null segment selector is a useful method of detecting accesses to unused segment registers and/or preventing unwanted accesses to data segments.

6.4.1.1 NULL Segment Checking in 64-bit Mode

In 64-bit mode, the processor does not perform runtime checking on NULL segment selectors. The processor does not cause a #GP fault when an attempt is made to access memory where the referenced segment register has a NULL segment selector.

6.5 PRIVILEGE LEVELS

The processor's segment-protection mechanism recognizes 4 privilege levels, numbered from 0 to 3. The greater numbers mean lesser privileges. Figure 6-3 shows how these levels of privilege can be interpreted as rings of protection.

The center (reserved for the most privileged code, data, and stacks) is used for the segments containing the critical software, usually the kernel of an operating system. Outer rings are used for less critical software. (Systems that use only 2 of the 4 possible privilege levels should use levels 0 and 3.)

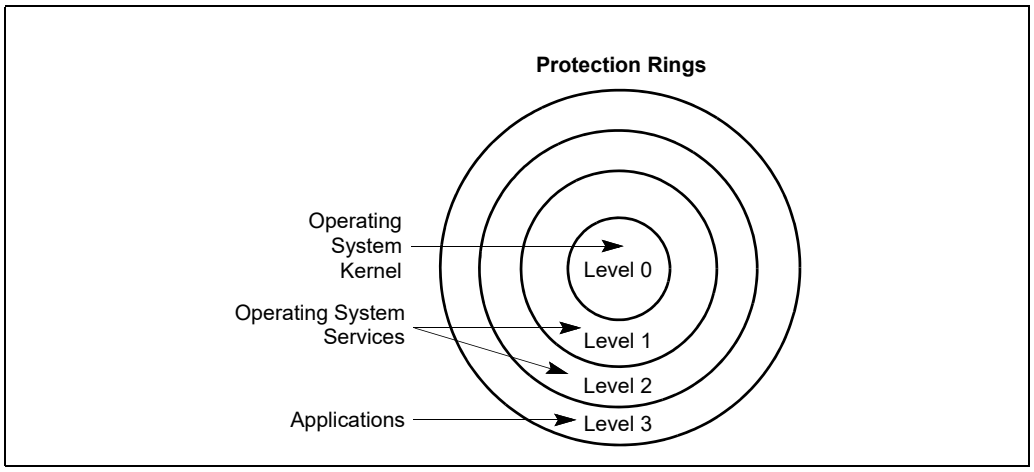


Figure 6-3. Protection Rings

The processor uses privilege levels to prevent a program or task operating at a lesser privilege level from accessing a segment with a greater privilege, except under controlled situations. When the processor detects a privilege level violation, it generates a general-protection exception (#GP).

To carry out privilege-level checks between code segments and data segments, the processor recognizes the following three types of privilege levels:

- **Current privilege level (CPL)** — The CPL is the privilege level of the currently executing program or task. It is stored in bits 0 and 1 of the CS and SS segment registers. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. The processor changes the CPL when program control is transferred to a code segment with a different privilege level. The CPL is treated slightly differently when accessing conforming code segments. Conforming code segments can be accessed from any privilege level that is equal to or numerically greater (less privileged) than the DPL of the conforming code segment. Also, the CPL is not changed when the processor accesses a conforming code segment that has a different privilege level than the CPL.
- **Descriptor privilege level (DPL)** — The DPL is the privilege level of a segment or gate. It is stored in the DPL field of the segment or gate descriptor for the segment or gate. When the currently executing code segment attempts to access a segment or gate, the DPL of the segment or gate is compared to the CPL and RPL of the segment or gate selector (as described later in this section). The DPL is interpreted differently, depending on the type of segment or gate being accessed:
  - **Data segment** — The DPL indicates the numerically highest privilege level that a program or task can have to be allowed to access the segment. For example, if the DPL of a data segment is 1, only programs running at a CPL of 0 or 1 can access the segment.
  - **Nonconforming code segment (without using a call gate)** — The DPL indicates the privilege level that a program or task must be at to access the segment. For example, if the DPL of a nonconforming code segment is 0, only programs running at a CPL of 0 can access the segment.
  - **Call gate** — The DPL indicates the numerically highest privilege level that the currently executing program or task can be at and still be able to access the call gate. (This is the same access rule as for a data segment.)
  - **Conforming code segment and nonconforming code segment accessed through a call gate** — The DPL indicates the numerically lowest privilege level that a program or task can have to be allowed to access the segment. For example, if the DPL of a conforming code segment is 2, programs running at a CPL of 0 or 1 cannot access the segment.

  - **TSS** — The DPL indicates the numerically highest privilege level that the currently executing program or task can be at and still be able to access the TSS. (This is the same access rule as for a data segment.)

- **Requested privilege level (RPL)** — The RPL is an override privilege level that is assigned to segment selectors. It is stored in bits 0 and 1 of the segment selector. The processor checks the RPL along with the CPL to determine if access to a segment is allowed. Even if the program or task requesting access to a segment has sufficient privilege to access the segment, access is denied if the RPL is not of sufficient privilege level. That is, if the RPL of a segment selector is numerically greater than the CPL, the RPL overrides the CPL, and vice versa. The RPL can be used to ensure that privileged code does not access a segment on behalf of an application program unless the program itself has access privileges for that segment. See Section 6.10.4, "Checking Caller Access Privileges (ARPL Instruction)," for a detailed description of the purpose and typical use of the RPL.

Privilege levels are checked when the segment selector of a segment descriptor is loaded into a segment register. The checks used for data access differ from those used for transfers of program control among code segments; therefore, the two kinds of accesses are considered separately in the following sections.

6.6 PRIVILEGE LEVEL CHECKING WHEN ACCESSING DATA SEGMENTS

To access operands in a data segment, the segment selector for the data segment must be loaded into the data-segment registers (DS, ES, FS, or GS) or into the stack-segment register (SS). (Segment registers can be loaded with the MOV, POP, LDS, LES, LFS, LGS, and LSS instructions.) Before the processor loads a segment selector into a segment register, it performs a privilege check (see Figure 6-4) by comparing the privilege levels of the currently running program or task (the CPL), the RPL of the segment selector, and the DPL of the segment's segment descriptor. The processor loads the segment selector into the segment register if the DPL is numerically greater than or equal to both the CPL and the RPL. Otherwise, a general-protection fault is generated and the segment register is not loaded.

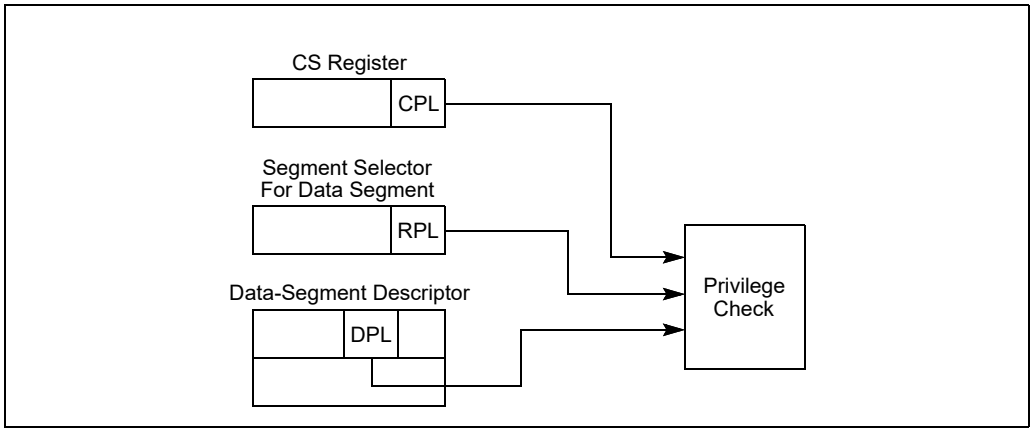


Figure 6-4. Privilege Check for Data Access

Figure 6-5 shows four procedures (located in codes segments A, B, C, and D), each running at different privilege levels and each attempting to access the same data segment.

1. The procedure in code segment A is able to access data segment E using segment selector E1, because the CPL of code segment A and the RPL of segment selector E1 are equal to the DPL of data segment E.
2. The procedure in code segment B is able to access data segment E using segment selector E2, because the CPL of code segment B and the RPL of segment selector E2 are both numerically lower than (more privileged) than the DPL of data segment E. A code segment B procedure can also access data segment E using segment selector E1.
3. The procedure in code segment C is not able to access data segment E using segment selector E3 (dotted line), because the CPL of code segment C and the RPL of segment selector E3 are both numerically greater than (less privileged) than the DPL of data segment E. Even if a code segment C procedure were to use segment selector

E1 or E2, such that the RPL would be acceptable, it still could not access data segment E because its CPL is not privileged enough.

4. The procedure in code segment D should be able to access data segment E because code segment D's CPL is numerically less than the DPL of data segment E. However, the RPL of segment selector E3 (which the code segment D procedure is using to access data segment E) is numerically greater than the DPL of data segment E, so access is not allowed. If the code segment D procedure were to use segment selector E1 or E2 to access the data segment, access would be allowed.

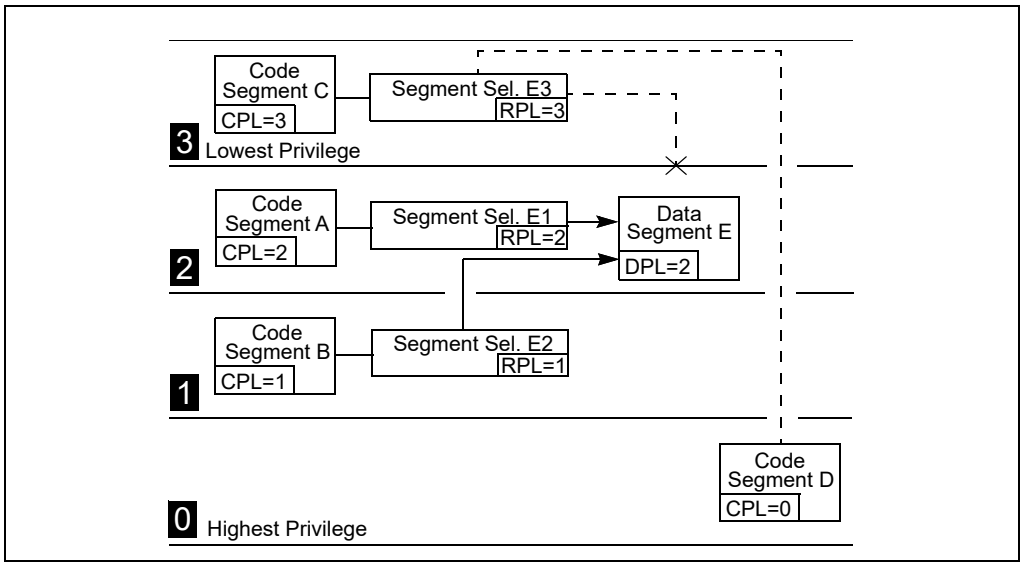


Figure 6-5. Examples of Accessing Data Segments From Various Privilege Levels

As demonstrated in the previous examples, the addressable domain of a program or task varies as its CPL changes. When the CPL is 0, data segments at all privilege levels are accessible; when the CPL is 1, only data segments at privilege levels 1 through 3 are accessible; when the CPL is 3, only data segments at privilege level 3 are accessible.

The RPL of a segment selector can always override the addressable domain of a program or task. When properly used, RPLs can prevent problems caused by accidental (or intensional) use of segment selectors for privileged data segments by less privileged programs or procedures.

It is important to note that the RPL of a segment selector for a data segment is under software control. For example, an application program running at a CPL of 3 can set the RPL for a data- segment selector to 0. With the RPL set to 0, only the CPL checks, not the RPL checks, will provide protection against deliberate, direct attempts to violate privilege-level security for the data segment. To prevent these types of privilege-level-check violations, a program or procedure can check access privileges whenever it receives a data-segment selector from another procedure (see Section 6.10.4, "Checking Caller Access Privileges (ARPL Instruction)").

6.6.1 Accessing Data in Code Segments

In some instances it may be desirable to access data structures that are contained in a code segment. The following methods of accessing data in code segments are possible:

- Load a data-segment register with a segment selector for a nonconforming, readable, code segment.
- Load a data-segment register with a segment selector for a conforming, readable, code segment.
- Use a code-segment override prefix (CS) to read a readable, code segment whose selector is already loaded in the CS register.

The same rules for accessing data segments apply to method 1. Method 2 is always valid because the privilege level of a conforming code segment is effectively the same as the CPL, regardless of its DPL. Method 3 is always valid because the DPL of the code segment selected by the CS register is the same as the CPL.

6.7 PRIVILEGE LEVEL CHECKING WHEN LOADING THE SS REGISTER

Privilege level checking also occurs when the SS register is loaded with the segment selector for a stack segment. Here all privilege levels related to the stack segment must match the CPL; that is, the CPL, the RPL of the stack-segment selector, and the DPL of the stack-segment descriptor must be the same. If the RPL and DPL are not equal to the CPL, a general-protection exception (#GP) is generated.

6.8 PRIVILEGE LEVEL CHECKING WHEN TRANSFERRING PROGRAM CONTROL BETWEEN CODE SEGMENTS

To transfer program control from one code segment to another, the segment selector for the destination code segment must be loaded into the code-segment register (CS). As part of this loading process, the processor examines the segment descriptor for the destination code segment and performs various limit, type, and privilege checks. If these checks are successful, the CS register is loaded, program control is transferred to the new code segment, and program execution begins at the instruction pointed to by the EIP register.

Program control transfers are carried out with the JMP, CALL, RET, SYSENTER, SYSEXIT, SYSCALL, SYSRET, INT *n*, and IRET instructions, as well as by the exception and interrupt mechanisms. Exceptions, interrupts, and the IRET instruction are special cases discussed in Chapter 7, "Interrupt and Exception Handling." This chapter discusses only the JMP, CALL, RET, SYSENTER, SYSEXIT, SYSCALL, and SYSRET instructions.

A JMP or CALL instruction can reference another code segment in any of four ways:

- The target operand contains the segment selector for the target code segment.
- The target operand points to a call-gate descriptor, which contains the segment selector for the target code segment.
- The target operand points to a TSS, which contains the segment selector for the target code segment.
- The target operand points to a task gate, which points to a TSS, which in turn contains the segment selector for the target code segment.

The following sections describe first two types of references. See Section 9.3, "Task Switching," for information on transferring program control through a task gate and/or TSS.

The SYSENTER and SYSEXIT instructions are special instructions for making fast calls to and returns from operating system or executive procedures. These instructions are discussed in Section 6.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."

The SYCALL and SYSRET instructions are special instructions for making fast calls to and returns from operating system or executive procedures in 64-bit mode. These instructions are discussed in Section 6.8.8, "Fast System Calls in 64-Bit Mode."

6.8.1 Direct Calls or Jumps to Code Segments

The near forms of the JMP, CALL, and RET instructions transfer program control within the current code segment, so privilege-level checks are not performed. The far forms of the JMP, CALL, and RET instructions transfer control to other code segments, so the processor does perform privilege-level checks.

When transferring program control to another code segment without going through a call gate, the processor examines four kinds of privilege level and type information (see Figure 6-6):

- The CPL. (Here, the CPL is the privilege level of the calling code segment; that is, the code segment that contains the procedure that is making the call or jump.)

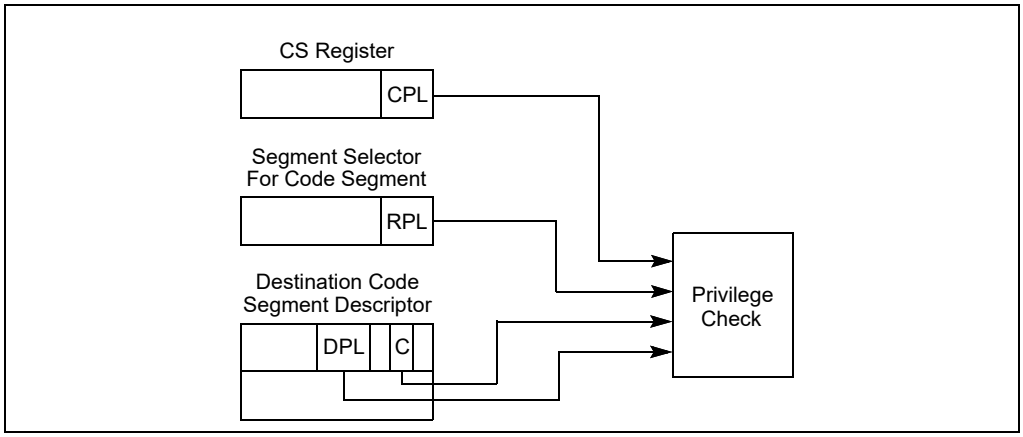


Figure 6-6. Privilege Check for Control Transfer Without Using a Gate

- The DPL of the segment descriptor for the destination code segment that contains the called procedure.
- The RPL of the segment selector of the destination code segment.
- The conforming (C) flag in the segment descriptor for the destination code segment, which determines whether the segment is a conforming (C flag is set) or nonconforming (C flag is clear) code segment. See Section 3.4.5.1, "Code- and Data-Segment Descriptor Types," for more information about this flag.

The rules that the processor uses to check the CPL, RPL, and DPL depends on the setting of the C flag, as described in the following sections.

6.8.1.1 Accessing Nonconforming Code Segments

When accessing nonconforming code segments, the CPL of the calling procedure must be equal to the DPL of the destination code segment; otherwise, the processor generates a general-protection exception (#GP). For example in Figure 6-7:

- Code segment C is a nonconforming code segment. A procedure in code segment A can call a procedure in code segment C (using segment selector C1) because they are at the same privilege level (CPL of code segment A is equal to the DPL of code segment C).
- A procedure in code segment B cannot call a procedure in code segment C (using segment selector C2 or C1) because the two code segments are at different privilege levels.

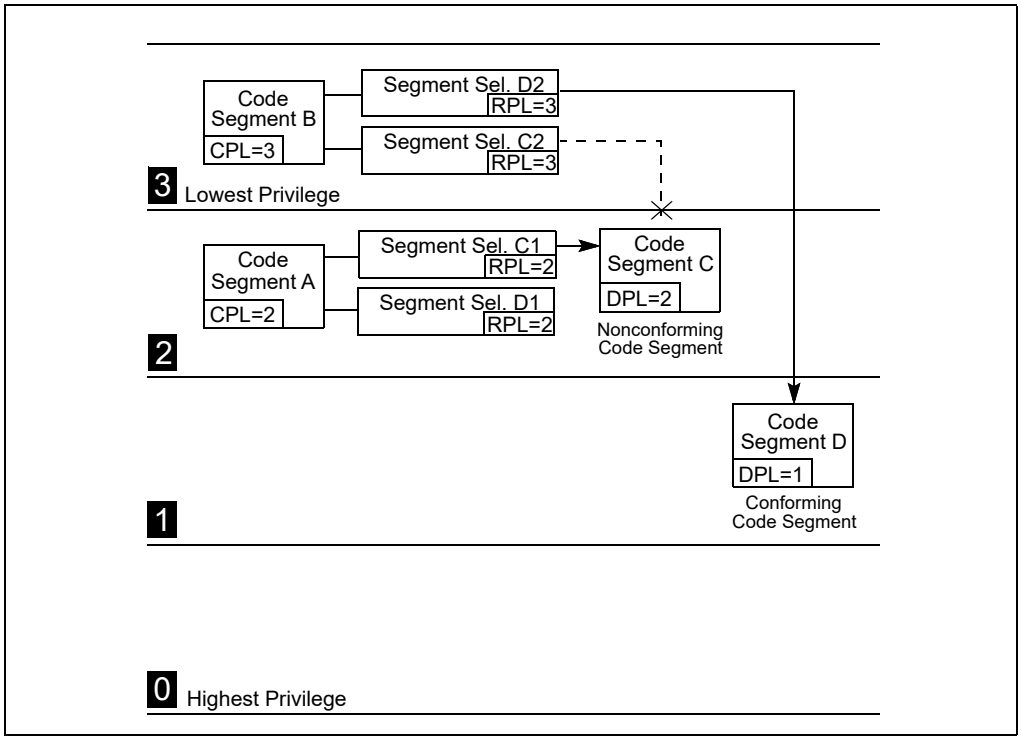


Figure 6-7. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels

The RPL of the segment selector that points to a nonconforming code segment has a limited effect on the privilege check. The RPL must be numerically less than or equal to the CPL of the calling procedure for a successful control transfer to occur. So, in the example in Figure 6-7, the RPLs of segment selectors C1 and C2 could legally be set to 0, 1, or 2, but not to 3.

When the segment selector of a nonconforming code segment is loaded into the CS register, the privilege level field is not changed; that is, it remains at the CPL (which is the privilege level of the calling procedure). This is true, even if the RPL of the segment selector is different from the CPL.

6.8.1.2 Accessing Conforming Code Segments

When accessing conforming code segments, the CPL of the calling procedure may be numerically equal to or greater than (less privileged) the DPL of the destination code segment; the processor generates a general-protection exception (#GP) only if the CPL is less than the DPL. (The segment selector RPL for the destination code segment is not checked if the segment is a conforming code segment.)

In the example in Figure 6-7, code segment D is a conforming code segment. Therefore, calling procedures in both code segment A and B can access code segment D (using either segment selector D1 or D2, respectively), because they both have CPLs that are greater than or equal to the DPL of the conforming code segment. **For conforming code segments, the DPL represents the numerically lowest privilege level that a calling procedure may be at to successfully make a call to the code segment.**

(Note that segments selectors D1 and D2 are identical except for their respective RPLs. But since RPLs are not checked when accessing conforming code segments, the two segment selectors are essentially interchangeable.)

When program control is transferred to a conforming code segment, the CPL does not change, even if the DPL of the destination code segment is less than the CPL. This situation is the only one where the CPL may be different from the DPL of the current code segment. Also, since the CPL does not change, no stack switch occurs.

Conforming segments are used for code modules such as math libraries and exception handlers, which support applications but do not require access to protected system facilities. These modules are part of the operating system or executive software, but they can be executed at numerically higher privilege levels (less privileged levels). Keeping the CPL at the level of a calling code segment when switching to a conforming code segment

prevents an application program from accessing nonconforming code segments while at the privilege level (DPL) of a conforming code segment and thus prevents it from accessing more privileged data.

Most code segments are nonconforming. For these segments, program control can be transferred only to code segments at the same level of privilege, unless the transfer is carried out through a call gate, as described in the following sections.

6.8.2 Gate Descriptors

To provide controlled access to code segments with different privilege levels, the processor provides special set of descriptors called gate descriptors. There are four kinds of gate descriptors:

- Call gates
- Trap gates
- Interrupt gates
- Task gates

Task gates are used for task switching and are discussed in Chapter 9, "Task Management." Trap and interrupt gates are special kinds of call gates used for calling exception and interrupt handlers. The are described in Chapter 7, "Interrupt and Exception Handling." This chapter is concerned only with call gates.

6.8.3 Call Gates

Call gates facilitate controlled transfers of program control between different privilege levels. They are typically used only in operating systems or executives that use the privilege-level protection mechanism. Call gates are also useful for transferring program control between 16-bit and 32-bit code segments, as described in Section 23.4, "Transferring Control Among Mixed-Size Code Segments."

Figure 6-8 shows the format of a call-gate descriptor. A call-gate descriptor may reside in the GDT or in an LDT, but not in the interrupt descriptor table (IDT). It performs six functions:

- It specifies the code segment to be accessed.
- It defines an entry point for a procedure in the specified code segment.
- It specifies the privilege level required for a caller trying to access the procedure.

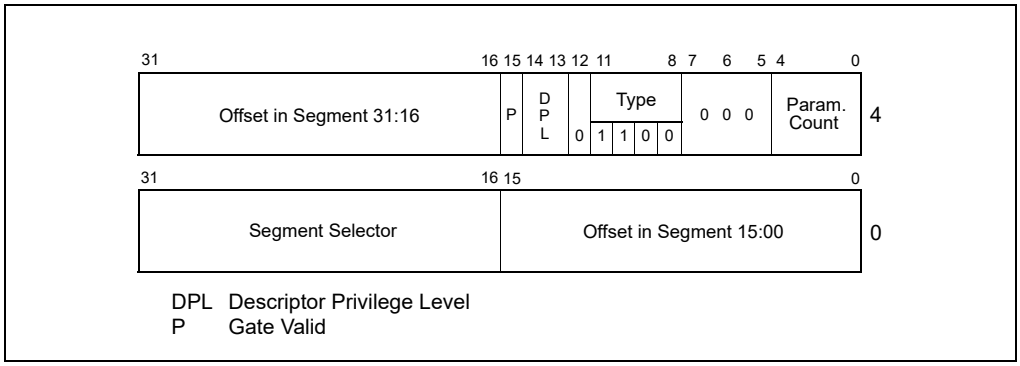


Figure 6-8. Call-Gate Descriptor

- If a stack switch occurs, it specifies the number of optional parameters to be copied between stacks.
- It defines the size of values to be pushed onto the target stack: 16-bit gates force 16-bit pushes and 32-bit gates force 32-bit pushes.
- It specifies whether the call-gate descriptor is valid.

The segment selector field in a call gate specifies the code segment to be accessed. The offset field specifies the entry point in the code segment. This entry point is generally to the first instruction of a specific procedure. The DPL field indicates the privilege level of the call gate, which in turn is the privilege level required to access the selected

procedure through the gate. The P flag indicates whether the call-gate descriptor is valid. (The presence of the code segment to which the gate points is indicated by the P flag in the code segment's descriptor.) The parameter count field indicates the number of parameters to copy from the calling procedures stack to the new stack if a stack switch occurs (see Section 6.8.5, "Stack Switching"). The parameter count specifies the number of words for 16-bit call gates and doublewords for 32-bit call gates.

Note that the P flag in a gate descriptor is normally always set to 1. If it is set to 0, a not present (#NP) exception is generated when a program attempts to access the descriptor. The operating system can use the P flag for special purposes. For example, it could be used to track the number of times the gate is used. Here, the P flag is initially set to 0 causing a trap to the not-present exception handler. The exception handler then increments a counter and sets the P flag to 1, so that on returning from the handler, the gate descriptor will be valid.

6.8.3.1 IA-32e Mode Call Gates

Call-gate descriptors in 32-bit mode provide a 32-bit offset for the instruction pointer (EIP); 64-bit extensions double the size of 32-bit mode call gates in order to store 64-bit instruction pointers (RIP). See Figure 6-9:

- The first eight bytes (bytes 7:0) of a 64-bit mode call gate are similar but not identical to legacy 32-bit mode call gates. The parameter-copy-count field has been removed.
- Bytes 11:8 hold the upper 32 bits of the target-segment offset in canonical form. A general-protection exception (#GP) is generated if software attempts to use a call gate with a target offset that is not in canonical form.
- 16-byte descriptors may reside in the same descriptor table with 16-bit and 32-bit descriptors. A type field, used for consistency checking, is defined in bits 12:8 of the 64-bit descriptor's highest dword (cleared to zero). A general-protection exception (#GP) results if an attempt is made to access the upper half of a 64-bit mode descriptor as a 32-bit mode descriptor.

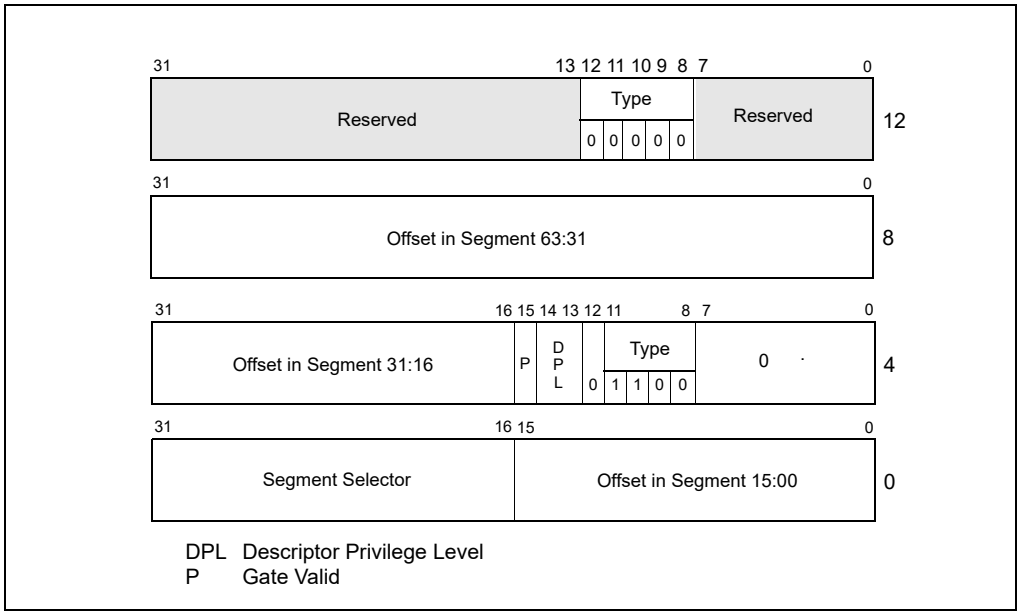


Figure 6-9. Call-Gate Descriptor in IA-32e Mode

- Target code segments referenced by a 64-bit call gate must be 64-bit code segments (CS.L = 1, CS.D = 0). If not, the reference generates a general-protection exception, #GP (CS selector).
- Only 64-bit mode call gates can be referenced in IA-32e mode (64-bit mode and compatibility mode). The legacy 32-bit mode call gate type (0CH) is redefined in IA-32e mode as a 64-bit call-gate type; no 32-bit call-gate type exists in IA-32e mode.

- If a far call references a 16-bit call gate type (04H) in IA-32e mode, a general-protection exception (#GP) is generated.

When a call references a 64-bit mode call gate, actions taken are identical to those taken in 32-bit mode, with the following exceptions:

- Stack pushes are made in eight-byte increments.
- A 64-bit RIP is pushed onto the stack.
- Parameter copying is not performed.

Use a matching far-return instruction size for correct operation (returns from 64-bit calls must be performed with a 64-bit operand-size return to process the stack correctly).

6.8.4 Accessing a Code Segment Through a Call Gate

To access a call gate, a far pointer to the gate is provided as a target operand in a CALL or JMP instruction. The segment selector from this pointer identifies the call gate (see Figure 6-10); the offset from the pointer is required, but not used or checked by the processor. (The offset can be set to any value.)

When the processor has accessed the call gate, it uses the segment selector from the call gate to locate the segment descriptor for the destination code segment. (This segment descriptor can be in the GDT or the LDT.) It then combines the base address from the code-segment descriptor with the offset from the call gate to form the linear address of the procedure entry point in the code segment.

As shown in Figure 6-11, four different privilege levels are used to check the validity of a program control transfer through a call gate:

- The CPL (current privilege level).
- The RPL (requestor privilege level) of the call gate's selector.
- The DPL (descriptor privilege level) of the call gate descriptor.
- The DPL of the segment descriptor of the destination code segment.

The C flag (conforming) in the segment descriptor for the destination code segment is also checked.

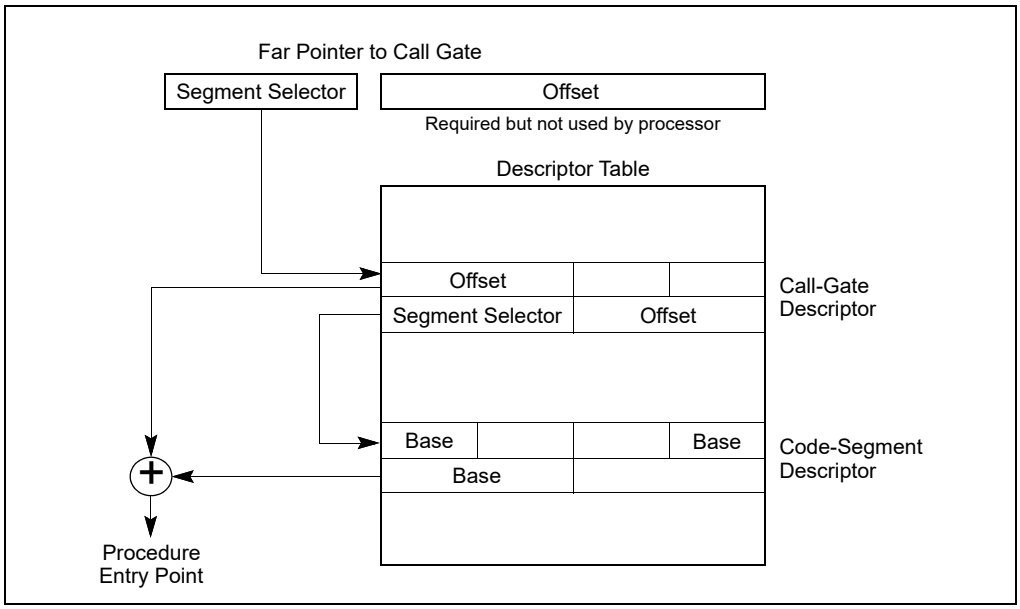


Figure 6-10. Call-Gate Mechanism

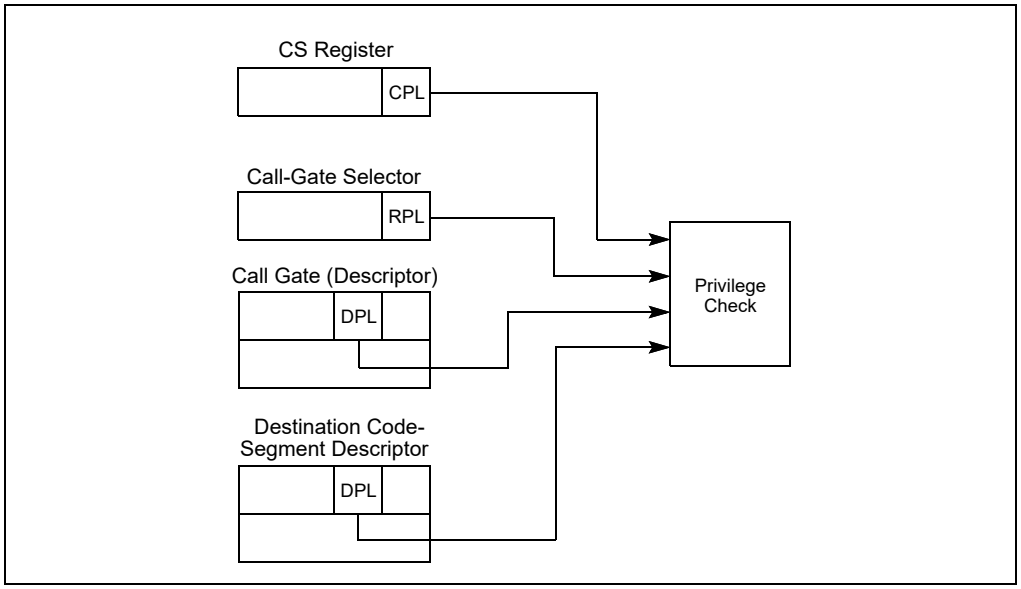


Figure 6-11. Privilege Check for Control Transfer with Call Gate

The privilege checking rules are different depending on whether the control transfer was initiated with a CALL or a JMP instruction, as shown in Table 6-1.

Table 6-1. Privilege Check Rules for Call Gates

Instruction	Privilege Check Rules
CALL	CPL ≤ call gate DPL; RPL ≤ call gate DPL Destination conforming code segment DPL ≤ CPL Destination nonconforming code segment DPL ≤ CPL
JMP	CPL ≤ call gate DPL; RPL ≤ call gate DPL Destination conforming code segment DPL ≤ CPL Destination nonconforming code segment DPL = CPL

The DPL field of the call-gate descriptor specifies the numerically highest privilege level from which a calling procedure can access the call gate; that is, to access a call gate, the CPL of a calling procedure must be equal to or less than the DPL of the call gate. For example, in Figure 6-15, call gate A has a DPL of 3. So calling procedures at all CPLs (0 through 3) can access this call gate, which includes calling procedures in code segments A, B, and C. Call gate B has a DPL of 2, so only calling procedures at a CPL or 0, 1, or 2 can access call gate B, which includes calling procedures in code segments B and C. The dotted line shows that a calling procedure in code segment A cannot access call gate B.

The RPL of the segment selector to a call gate must satisfy the same test as the CPL of the calling procedure; that is, the RPL must be less than or equal to the DPL of the call gate. In the example in Figure 6-15, a calling procedure in code segment C can access call gate B using gate selector B2 or B1, but it could not use gate selector B3 to access call gate B.

If the privilege checks between the calling procedure and call gate are successful, the processor then checks the DPL of the code-segment descriptor against the CPL of the calling procedure. Here, the privilege check rules vary between CALL and JMP instructions. Only CALL instructions can use call gates to transfer program control to more privileged (numerically lower privilege level) nonconforming code segments; that is, to nonconforming code segments with a DPL less than the CPL. A JMP instruction can use a call gate only to transfer program control to a nonconforming code segment with a DPL equal to the CPL. CALL and JMP instruction can both transfer program control to a more privileged conforming code segment; that is, to a conforming code segment with a DPL less than or equal to the CPL.

If a call is made to a more privileged (numerically lower privilege level) nonconforming destination code segment, the CPL is lowered to the DPL of the destination code segment and a stack switch occurs (see Section 6.8.5, "Stack Switching"). If a call or jump is made to a more privileged conforming destination code segment, the CPL is not changed and no stack switch occurs.

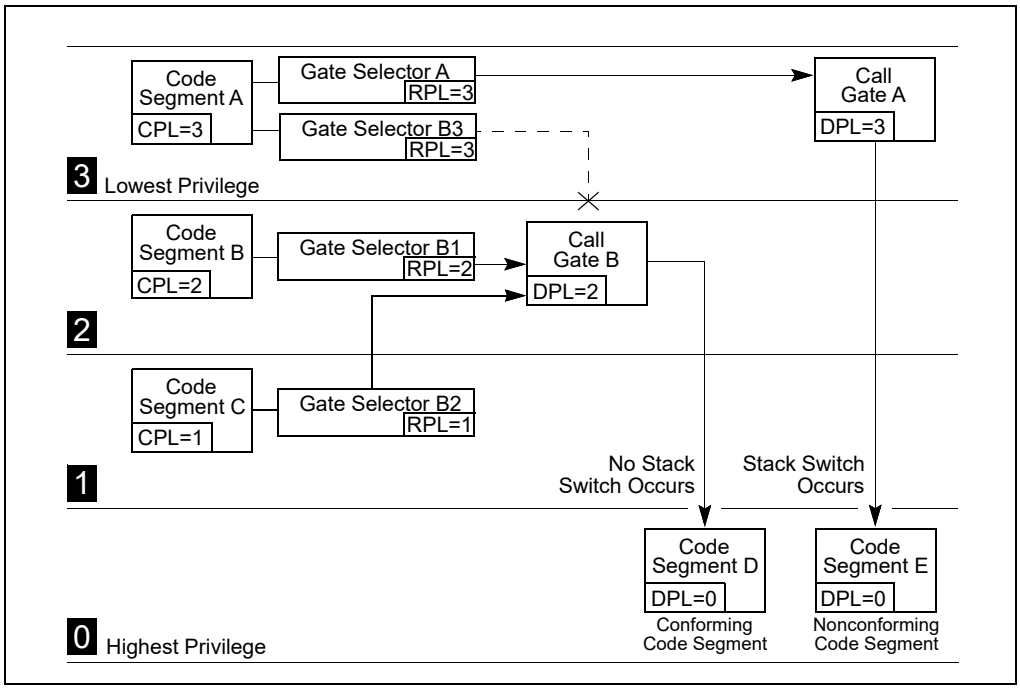


Figure 6-12. Example of Accessing Call Gates At Various Privilege Levels

Call gates allow a single code segment to have procedures that can be accessed at different privilege levels. For example, an operating system located in a code segment may have some services which are intended to be used by both the operating system and application software (such as procedures for handling character I/O). Call gates for these procedures can be set up that allow access at all privilege levels (0 through 3). More privileged call gates (with DPLs of 0 or 1) can then be set up for other operating system services that are intended to be used only by the operating system (such as procedures that initialize device drivers).

6.8.5 Stack Switching

Whenever a call gate is used to transfer program control to a more privileged nonconforming code segment (that is, when the DPL of the nonconforming destination code segment is less than the CPL), the processor automatically switches to the stack for the destination code segment's privilege level. This stack switching is carried out to prevent more privileged procedures from crashing due to insufficient stack space. It also prevents less privileged procedures from interfering (by accident or intent) with more privileged procedures through a shared stack.

Each task must define up to 4 stacks: one for applications code (running at privilege level 3) and one for each of the privilege levels 2, 1, and 0 that are used. (If only two privilege levels are used [3 and 0], then only two stacks must be defined.) Each of these stacks is located in a separate segment and is identified with a segment selector and an offset into the stack segment (a stack pointer).

The segment selector and stack pointer for the privilege level 3 stack is located in the SS and ESP registers, respectively, when privilege-level-3 code is being executed and is automatically stored on the called procedure's stack when a stack switch occurs.

Pointers to the privilege level 0, 1, and 2 stacks are stored in the TSS for the currently running task (see Figure 9-2). Each of these pointers consists of a segment selector and a stack pointer (loaded into the ESP register). These initial pointers are strictly read-only values. The processor does not change them while the task is running. They are used only to create new stacks when calls are made to more privileged levels (numerically lower

privilege levels). These stacks are disposed of when a return is made from the called procedure. The next time the procedure is called, a new stack is created using the initial stack pointer. (The TSS does not specify a stack for privilege level 3 because the processor does not allow a transfer of program control from a procedure running at a CPL of 0, 1, or 2 to a procedure running at a CPL of 3, except on a return.)

The operating system is responsible for creating stacks and stack-segment descriptors for all the privilege levels to be used and for loading initial pointers for these stacks into the TSS. Each stack must be read/write accessible (as specified in the type field of its segment descriptor) and must contain enough space (as specified in the limit field) to hold the following items:

- The contents of the SS, ESP, CS, and EIP registers for the calling procedure.
- The parameters and temporary variables required by the called procedure.
- The EFLAGS register and error code, when implicit calls are made to an exception or interrupt handler.

The stack will need to require enough space to contain many frames of these items, because procedures often call other procedures, and an operating system may support nesting of multiple interrupts. Each stack should be large enough to allow for the worst case nesting scenario at its privilege level.

(If the operating system does not use the processor's multitasking mechanism, it still must create at least one TSS for this stack-related purpose.)

When a procedure call through a call gate results in a change in privilege level, the processor performs the following steps to switch stacks and begin execution of the called procedure at a new privilege level:

1. Uses the DPL of the destination code segment (the new CPL) to select a pointer to the new stack (segment selector and stack pointer) from the TSS.
2. Reads the segment selector and stack pointer for the stack to be switched to from the current TSS. Any limit violations detected while reading the stack-segment selector, stack pointer, or stack-segment descriptor cause an invalid TSS (#TS) exception to be generated.
3. Checks the stack-segment descriptor for the proper privileges and type and generates an invalid TSS (#TS) exception if violations are detected.
4. Temporarily saves the current values of the SS and ESP registers.
5. Loads the segment selector and stack pointer for the new stack in the SS and ESP registers.
6. Pushes the temporarily saved values for the SS and ESP registers (for the calling procedure) onto the new stack (see Figure 6-13).
7. Copies the number of parameter specified in the parameter count field of the call gate from the calling procedure's stack to the new stack. If the count is 0, no parameters are copied.
8. Pushes the return instruction pointer (the current contents of the CS and EIP registers) onto the new stack.
9. Loads the segment selector for the new code segment and the new instruction pointer from the call gate into the CS and EIP registers, respectively, and begins execution of the called procedure.

See the description of the CALL instruction in Chapter 3, "Instruction Set Reference, A-L,", in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, for a detailed description of the privilege level checks and other protection checks that the processor performs on a far call through a call gate.

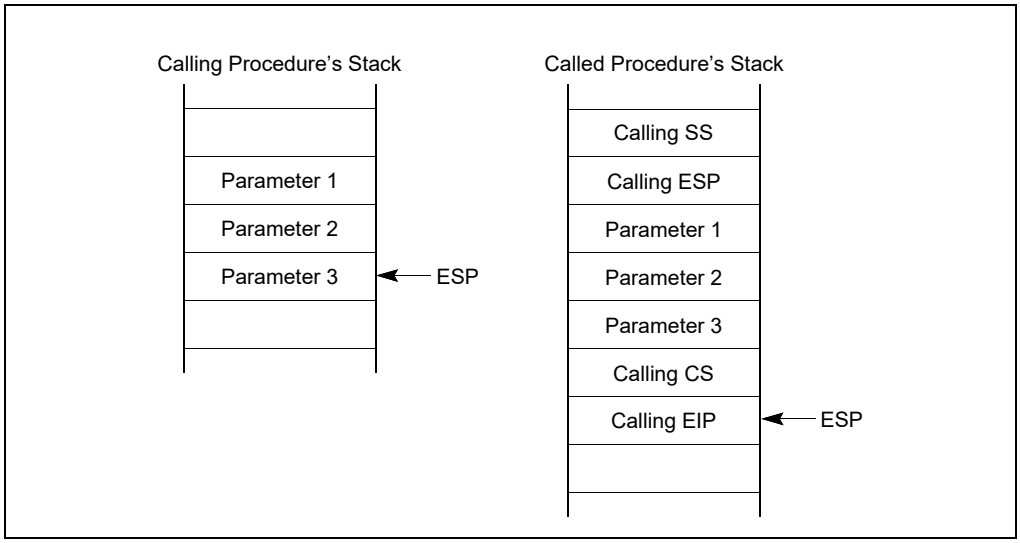


Figure 6-13. Stack Switching During an Interprivilege-Level Call

The parameter count field in a call gate specifies the number of data items (up to 31) that the processor should copy from the calling procedure's stack to the stack of the called procedure. If more than 31 data items need to be passed to the called procedure, one of the parameters can be a pointer to a data structure, or the saved contents of the SS and ESP registers may be used to access parameters in the old stack space. The size of the data items passed to the called procedure depends on the call gate size, as described in Section 6.8.3, "Call Gates."

6.8.5.1 Stack Switching in 64-bit Mode

Although protection-check rules for call gates are unchanged from 32-bit mode, stack-switch changes in 64-bit mode are different.

When stacks are switched as part of a 64-bit mode privilege-level change through a call gate, a new SS (stack segment) descriptor is not loaded; 64-bit mode only loads an inner-level RSP from the TSS. The new SS is forced to NULL and the SS selector's RPL field is forced to the new CPL. The new SS is set to NULL in order to handle nested far transfers (far CALL, INTn, interrupts, and exceptions). The old SS and RSP are saved on the new stack.

On a subsequent far RET, the old SS is popped from the stack and loaded into the SS register. See Table 6-2.

Table 6-2. 64-Bit-Mode Stack Layout After Far CALL with CPL Change

32-bit Mode			IA-32e mode		
Old SS Selector	+12		+24	Old SS Selector	
Old ESP	+8		+16	Old RSP	
CS Selector	+4		+8	Old CS Selector	
EIP	0	**ESP**	0	RIP	**RSP**
< 4 Bytes >				< 8 Bytes >	

In 64-bit mode, stack operations resulting from a privilege-level-changing far call or far return are eight-bytes wide and change the RSP by eight. The mode does not support the automatic parameter-copy feature found in 32-bit mode. The call-gate count field is ignored. Software can access the old stack, if necessary, by referencing the old stack-segment selector and stack pointer saved on the new process stack.

In 64-bit mode, far RET is allowed to load a NULL SS under certain conditions. If the target mode is 64-bit mode and the target CPL ≠ 3, IRET allows SS to be loaded with a NULL selector. If the called procedure itself is interrupted, the NULL SS is pushed on the stack frame. On the subsequent far RET, the NULL SS on the stack acts as a flag to tell the processor not to load a new SS descriptor.

6.8.6 Returning from a Called Procedure

The RET instruction can be used to perform a near return, a far return at the same privilege level, and a far return to a different privilege level. This instruction is intended to execute returns from procedures that were called with a CALL instruction. It does not support returns from a JMP instruction, because the JMP instruction does not save a return instruction pointer on the stack.

A near return only transfers program control within the current code segment; therefore, the processor performs only a limit check. When the processor pops the return instruction pointer from the stack into the EIP register, it checks that the pointer does not exceed the limit of the current code segment.

On a far return at the same privilege level, the processor pops both a segment selector for the code segment being returned to and a return instruction pointer from the stack. Under normal conditions, these pointers should be valid, because they were pushed on the stack by the CALL instruction. However, the processor performs privilege checks to detect situations where the current procedure might have altered the pointer or failed to maintain the stack properly.

A far return that requires a privilege-level change is only allowed when returning to a less privileged level (that is, the DPL of the return code segment is numerically greater than the CPL). The processor uses the RPL field from the CS register value saved for the calling procedure (see Figure 6-13) to determine if a return to a numerically higher privilege level is required. If the RPL is numerically greater (less privileged) than the CPL, a return across privilege levels occurs.

The processor performs the following steps when performing a far return to a calling procedure (see Figures 6-2 and 6-4 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for an illustration of the stack contents prior to and after a return):

1. Checks the RPL field of the saved CS register value to determine if a privilege level change is required on the return.
2. Loads the CS and EIP registers with the values on the called procedure's stack. (Type and privilege level checks are performed on the code-segment descriptor and RPL of the code- segment selector.)
3. (If the RET instruction includes a parameter count operand and the return requires a privilege level change.) Adds the parameter count (in bytes obtained from the RET instruction) to the current ESP register value (after popping the CS and EIP values), to step past the parameters on the called procedure's stack. The resulting value in the ESP register points to the saved SS and ESP values for the calling procedure's stack. (Note that the byte count in the RET instruction must be chosen to match the parameter count in the call gate that the calling procedure referenced when it made the original call multiplied by the size of the parameters.)
4. (If the return requires a privilege level change.) Loads the SS and ESP registers with the saved SS and ESP values and switches back to the calling procedure's stack. The SS and ESP values for the called procedure's stack are discarded. Any limit violations detected while loading the stack-segment selector or stack pointer cause a general-protection exception (#GP) to be generated. The new stack-segment descriptor is also checked for type and privilege violations.
5. (If the RET instruction includes a parameter count operand.) Adds the parameter count (in bytes obtained from the RET instruction) to the current ESP register value, to step past the parameters on the calling procedure's stack. The resulting ESP value is not checked against the limit of the stack segment. If the ESP value is beyond the limit, that fact is not recognized until the next stack operation.
6. (If the return requires a privilege level change.) Checks the contents of the DS, ES, FS, and GS segment registers. If any of these registers refer to segments whose DPL is less than the new CPL (excluding conforming code segments), the segment register is loaded with a null segment selector.

See the description of the RET instruction in Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, for a detailed description of the privilege level checks and other protection checks that the processor performs on a far return.

6.8.7 Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processors for the purpose of providing a fast (low overhead) mechanism for calling operating system or executive procedures.

SYSENTER is intended for use by user code running at privilege level 3 to access operating system or executive procedures running at privilege level 0. SYSEXIT is intended for use by privilege level 0 operating system or executive procedures for fast returns to privilege level 3 user code. SYSENTER can be executed from privilege levels 3, 2, 1, or 0; SYSEXIT can only be executed from privilege level 0.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. This is because SYSENTER does not save any state information for use by SYSEXIT on a return.

The target instruction and stack pointer for these instructions are not specified through instruction operands. Instead, they are specified through parameters entered in MSRs and general-purpose registers.

For SYSENTER, target fields are generated using the following sources:

- **Target code segment** — Reads this from IA32_SYSENTER_CS.
- **Target instruction** — Reads this from IA32_SYSENTER_EIP.
- **Stack segment** — Computed by adding 8 to the value in IA32_SYSENTER_CS.
- **Stack pointer** — Reads this from the IA32_SYSENTER_ESP.

For SYSEXIT, target fields are generated using the following sources:

- **Target code segment** — Computed by adding 16 to the value in the IA32_SYSENTER_CS.
- **Target instruction** — Reads this from EDX.
- **Stack segment** — Computed by adding 24 to the value in IA32_SYSENTER_CS.
- **Stack pointer** — Reads this from ECX.

The SYSENTER and SYSEXIT instructions perform "fast" calls and returns because they force the processor into a predefined privilege level 0 state when SYSENTER is executed and into a predefined privilege level 3 state when SYSEXIT is executed. By forcing predefined and consistent processor states, the number of privilege checks ordinarily required to perform a far call to another privilege levels are greatly reduced. Also, by predefining the target context state in MSRs and general-purpose registers eliminates all memory accesses except when fetching the target code.

Any additional state that needs to be saved to allow a return to the calling procedure must be saved explicitly by the calling procedure or be predefined through programming conventions.

6.8.7.1 SYSENTER and SYSEXIT Instructions in IA-32e Mode

For Intel 64 processors, the SYSENTER and SYSEXIT instructions are enhanced to allow fast system calls from user code running at privilege level 3 (in compatibility mode or 64-bit mode) to 64-bit executive procedures running at privilege level 0. IA32_SYSENTER_EIP MSR and IA32_SYSENTER_ESP MSR are expanded to hold 64-bit addresses. If IA-32e mode is inactive, only the lower 32-bit addresses stored in these MSRs are used. The WRMSR instruction ensures that the addresses stored in these MSRs are canonical. Note that, in 64-bit mode, IA32_SYSENTER_CS must not contain a NULL selector.

When SYSENTER transfers control, the following fields are generated and bits set:

- **Target code segment** — Reads non-NULL selector from IA32_SYSENTER_CS.
- **New CS attributes** — CS base = 0, CS limit = FFFFFFFFH.
- **Target instruction** — Reads 64-bit canonical address from IA32_SYSENTER_EIP.
- **Stack segment** — Computed by adding 8 to the value from IA32_SYSENTER_CS.
- **Stack pointer** — Reads 64-bit canonical address from IA32_SYSENTER_ESP.
- **New SS attributes** — SS base = 0, SS limit = FFFFFFFFH.

When the SYSEXIT instruction transfers control to 64-bit mode user code using REX.W, the following fields are generated and bits set:

- **Target code segment** — Computed by adding 32 to the value in IA32_SYSENTER_CS.
- **New CS attributes** — L-bit = 1 (go to 64-bit mode).
- **Target instruction** — Reads 64-bit canonical address in RDX.
- **Stack segment** — Computed by adding 40 to the value of IA32_SYSENTER_CS.

- **Stack pointer** — Update RSP using 64-bit canonical address in RCX.

When SYSEXIT transfers control to compatibility mode user code when the operand size attribute is 32 bits, the following fields are generated and bits set:

- **Target code segment** — Computed by adding 16 to the value in IA32_SYSENTER_CS.
- **New CS attributes** — L-bit = 0 (go to compatibility mode).
- **Target instruction** — Fetch the target instruction from 32-bit address in EDX.
- **Stack segment** — Computed by adding 24 to the value in IA32_SYSENTER_CS.
- **Stack pointer** — Update ESP from 32-bit address in ECX.

6.8.8 Fast System Calls in 64-Bit Mode

The SYSCALL and SYSRET instructions are designed for operating systems that use a flat memory model (segmentation is not used). The instructions, along with SYSENTER and SYSEXIT, are suited for IA-32e mode operation. SYSCALL and SYSRET, however, are not supported in compatibility mode (or in protected mode). Use CPUID to check if SYSCALL and SYSRET are available (CPUID.80000001H.EDX[bit 11] = 1).

SYSCALL is intended for use by user code running at privilege level 3 to access operating system or executive procedures running at privilege level 0. SYSRET is intended for use by privilege level 0 operating system or executive procedures for fast returns to privilege level 3 user code.

Stack pointers for SYSCALL/SYSRET are not specified through model specific registers. The clearing of bits in RFLAGS is programmable rather than fixed. SYSCALL/SYSRET save and restore the RFLAGS register.

For SYSCALL, the processor saves RFLAGS into R11 and the RIP of the next instruction into RCX; it then gets the privilege-level 0 target code segment, instruction pointer, stack segment, and flags as follows:

- **Target code segment** — Reads a non-NULL selector from IA32_STAR[47:32].
- **Target instruction pointer** — Reads a 64-bit address from IA32_LSTAR. (The WRMSR instruction ensures that the value of the IA32_LSTAR MSR is canonical.)
- **Stack segment** — Computed by adding 8 to the value in IA32_STAR[47:32].
- **Flags** — The processor sets RFLAGS to the logical-AND of its current value with the complement of the value in the IA32_FMASK MSR.

When SYSRET transfers control to 64-bit mode user code using REX.W, the processor gets the privilege level 3 target code segment, instruction pointer, stack segment, and flags as follows:

- **Target code segment** — Reads a non-NULL selector from IA32_STAR[63:48] + 16.
- **Target instruction pointer** — Copies the value in RCX into RIP.
- **Stack segment** — IA32_STAR[63:48] + 8.
- **EFLAGS** — Loaded from R11.

When SYSRET transfers control to 32-bit mode user code using a 32-bit operand size, the processor gets the privilege level 3 target code segment, instruction pointer, stack segment, and flags as follows:

- **Target code segment** — Reads a non-NULL selector from IA32_STAR[63:48].
- **Target instruction pointer** — Copies the value in ECX into EIP.
- **Stack segment** — IA32_STAR[63:48] + 8.
- **EFLAGS** — Loaded from R11.

It is the responsibility of the OS to ensure the descriptors in the GDT/LDT correspond to the selectors loaded by SYSCALL/SYSRET (consistent with the base, limit, and attribute values forced by the instructions).

See Figure 6-14 for the layout of IA32_STAR, IA32_LSTAR, and IA32_FMASK.

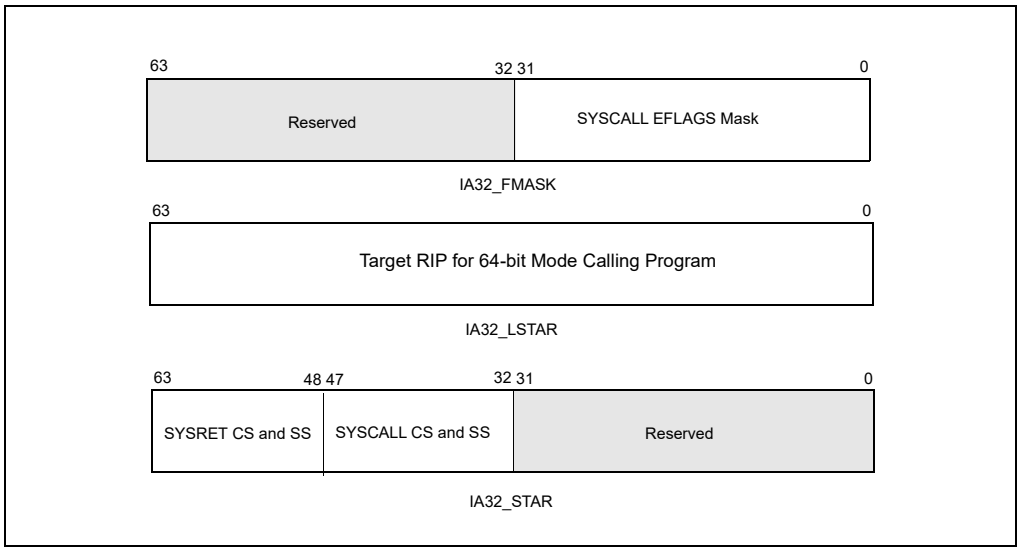


Figure 6-14. MSRs Used by SYSCALL and SYSRET

The SYSCALL instruction does not save the stack pointer, and the SYSRET instruction does not restore it. It is likely that the OS system-call handler will change the stack pointer from the user stack to the OS stack. If so, it is the responsibility of software first to save the user stack pointer. This might be done by user code, prior to executing SYSCALL, or by the OS system-call handler after SYSCALL.

Because the SYSRET instruction does not modify the stack pointer, it is necessary for software to switch back to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.

If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following:

- External interrupts. The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.
- Nonmaskable interrupts (NMIs). The OS can ensure that the NMI handler is invoked with the correct stack by using the interrupt stack table (IST) mechanism for gate 2 (NMI) in the IDT (see Section 7.14.5, "Interrupt Stack Table").
- General-protection exceptions (#GP). The SYSRET instruction generates #GP(0) if the value of RCX is not canonical. The OS can address this possibility using one or more of the following approaches:
  - — Confirming that the value of RCX is canonical before executing SYSRET.
  - — Using paging to ensure that the SYSCALL instruction will never save a non-canonical value into RCX.
  - — Using the IST mechanism for gate 13 (#GP) in the IDT.

6.9 PRIVILEGED INSTRUCTIONS

Some of the system instructions (called "privileged instructions") are protected from use by application programs. The privileged instructions control system functions (such as the loading of system registers). They can be executed only when the CPL is 0 (most privileged). If one of these instructions is executed when the CPL is not 0, a general-protection exception (#GP) is generated. The following system instructions are privileged instructions:

- LGDT — Load GDT register.
- LLDT — Load LDT register.

- LTR — Load task register.
- LIDT — Load IDT register.
- MOV (control registers) — Load and store control registers.
- LMSW — Load machine status word.
- CLTS — Clear task-switched flag in register CR0.
- MOV (debug registers) — Load and store debug registers.
- INVD — Invalidate cache, without writeback.
- WBINVD — Invalidate cache, with writeback.
- INVLPG —Invalidate TLB entry.
- HLT— Halt processor.
- RDMSR — Read Model-Specific Registers.
- WRMSR —Write Model-Specific Registers.
- RDPMC — Read Performance-Monitoring Counter.
- RDTSC — Read Time-Stamp Counter.

Some of the privileged instructions are available only in the more recent families of Intel 64 and IA-32 processors (see Section 24.13, "New Instructions In the Pentium and Later IA-32 Processors").

The PCE and TSD flags in register CR4 (bits 4 and 2, respectively) enable the RDPMC and RDTSC instructions, respectively, to be executed at any CPL.

6.10 POINTER VALIDATION

When operating in protected mode, the processor validates all pointers to enforce protection between segments and maintain isolation between privilege levels. Pointer validation consists of the following checks:

1. Checking access rights to determine if the segment type is compatible with its use.
2. Checking read/write rights.
3. Checking if the pointer offset exceeds the segment limit.
4. Checking if the supplier of the pointer is allowed to access the segment.
5. Checking the offset alignment.

The processor automatically performs first, second, and third checks during instruction execution. Software must explicitly request the fourth check by issuing an ARPL instruction. The fifth check (offset alignment) is performed automatically at privilege level 3 if alignment checking is turned on. Offset alignment does not affect isolation of privilege levels.

6.10.1 Checking Access Rights (LAR Instruction)

When the processor accesses a segment using a far pointer, it performs an access rights check on the segment descriptor pointed to by the far pointer. This check is performed to determine if type and privilege level (DPL) of the segment descriptor are compatible with the operation to be performed. For example, when making a far call in protected mode, the segment-descriptor type must be for a conforming or nonconforming code segment, a call gate, a task gate, or a TSS. Then, if the call is to a nonconforming code segment, the DPL of the code segment must be equal to the CPL, and the RPL of the code segment's segment selector must be less than or equal to the DPL. If type or privilege level are found to be incompatible, the appropriate exception is generated.

To prevent type incompatibility exceptions from being generated, software can check the access rights of a segment descriptor using the LAR (load access rights) instruction. The LAR instruction specifies the segment selector for the segment descriptor whose access rights are to be checked and a destination register. The instruction then performs the following operations:

1. Check that the segment selector is not null.

2. Checks that the segment selector points to a segment descriptor that is within the descriptor table limit (GDT or LDT).
3. Checks that the segment descriptor is a code, data, LDT, call gate, task gate, or TSS segment-descriptor type.
4. If the segment is not a conforming code segment, checks if the segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL).
5. If the privilege level and type checks pass, loads the second doubleword of the segment descriptor into the destination register (masked by the value 00FXFF00H, where X indicates that the corresponding 4 bits are undefined) and sets the ZF flag in the EFLAGS register. If the segment selector is not visible at the current privilege level or is an invalid type for the LAR instruction, the instruction does not modify the destination register and clears the ZF flag.

Once loaded in the destination register, software can perform additional checks on the access rights information.

6.10.2 Checking Read/Write Rights (VERR and VERW Instructions)

When the processor accesses any code or data segment it checks the read/write privileges assigned to the segment to verify that the intended read or write operation is allowed. Software can check read/write rights using the VERR (verify for reading) and VERW (verify for writing) instructions. Both these instructions specify the segment selector for the segment being checked. The instructions then perform the following operations:

1. Check that the segment selector is not null.
2. Checks that the segment selector points to a segment descriptor that is within the descriptor table limit (GDT or LDT).
3. Checks that the segment descriptor is a code or data-segment descriptor type.
4. If the segment is not a conforming code segment, checks if the segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL).
5. Checks that the segment is readable (for the VERR instruction) or writable (for the VERW) instruction.

The VERR instruction sets the ZF flag in the EFLAGS register if the segment is visible at the CPL and readable; the VERW sets the ZF flag if the segment is visible and writable. (Code segments are never writable.) The ZF flag is cleared if any of these checks fail.

6.10.3 Checking That the Pointer Offset Is Within Limits (LSL Instruction)

When the processor accesses any segment it performs a limit check to ensure that the offset is within the limit of the segment. Software can perform this limit check using the LSL (load segment limit) instruction. Like the LAR instruction, the LSL instruction specifies the segment selector for the segment descriptor whose limit is to be checked and a destination register. The instruction then performs the following operations:

1. Check that the segment selector is not null.
2. Checks that the segment selector points to a segment descriptor that is within the descriptor table limit (GDT or LDT).
3. Checks that the segment descriptor is a code, data, LDT, or TSS segment-descriptor type.
4. If the segment is not a conforming code segment, checks if the segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector less than or equal to the DPL).
5. If the privilege level and type checks pass, loads the unscrambled limit (the limit scaled according to the setting of the G flag in the segment descriptor) into the destination register and sets the ZF flag in the EFLAGS register. If the segment selector is not visible at the current privilege level or is an invalid type for the LSL instruction, the instruction does not modify the destination register and clears the ZF flag.

Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

6.10.4 Checking Caller Access Privileges (ARPL Instruction)

The requestor's privilege level (RPL) field of a segment selector is intended to carry the privilege level of a calling procedure (the calling procedure's CPL) to a called procedure. The called procedure then uses the RPL to determine if access to a segment is allowed. The RPL is said to "weaken" the privilege level of the called procedure to that of the RPL.

Operating-system procedures typically use the RPL to prevent less privileged application programs from accessing data located in more privileged segments. When an operating-system procedure (the called procedure) receives a segment selector from an application program (the calling procedure), it sets the segment selector's RPL to the privilege level of the calling procedure. Then, when the operating system uses the segment selector to access its associated segment, the processor performs privilege checks using the calling procedure's privilege level (stored in the RPL) rather than the numerically lower privilege level (the CPL) of the operating-system procedure. The RPL thus ensures that the operating system does not access a segment on behalf of an application program unless that program itself has access to the segment.

Figure 6-15 shows an example of how the processor uses the RPL field. In this example, an application program (located in code segment A) possesses a segment selector (segment selector D1) that points to a privileged data structure (that is, a data structure located in a data segment D at privilege level 0).

The application program cannot access data segment D, because it does not have sufficient privilege, but the operating system (located in code segment C) can. So, in an attempt to access data segment D, the application program executes a call to the operating system and passes segment selector D1 to the operating system as a parameter on the stack. Before passing the segment selector, the (well behaved) application program sets the RPL of the segment selector to its current privilege level (which in this example is 3). If the operating system attempts to access data segment D using segment selector D1, the processor compares the CPL (which is now 0 following the call), the RPL of segment selector D1, and the DPL of data segment D (which is 0). Since the RPL is greater than the DPL, access to data segment D is denied. The processor's protection mechanism thus protects data segment D from access by the operating system, because application program's privilege level (represented by the RPL of segment selector B) is greater than the DPL of data segment D.

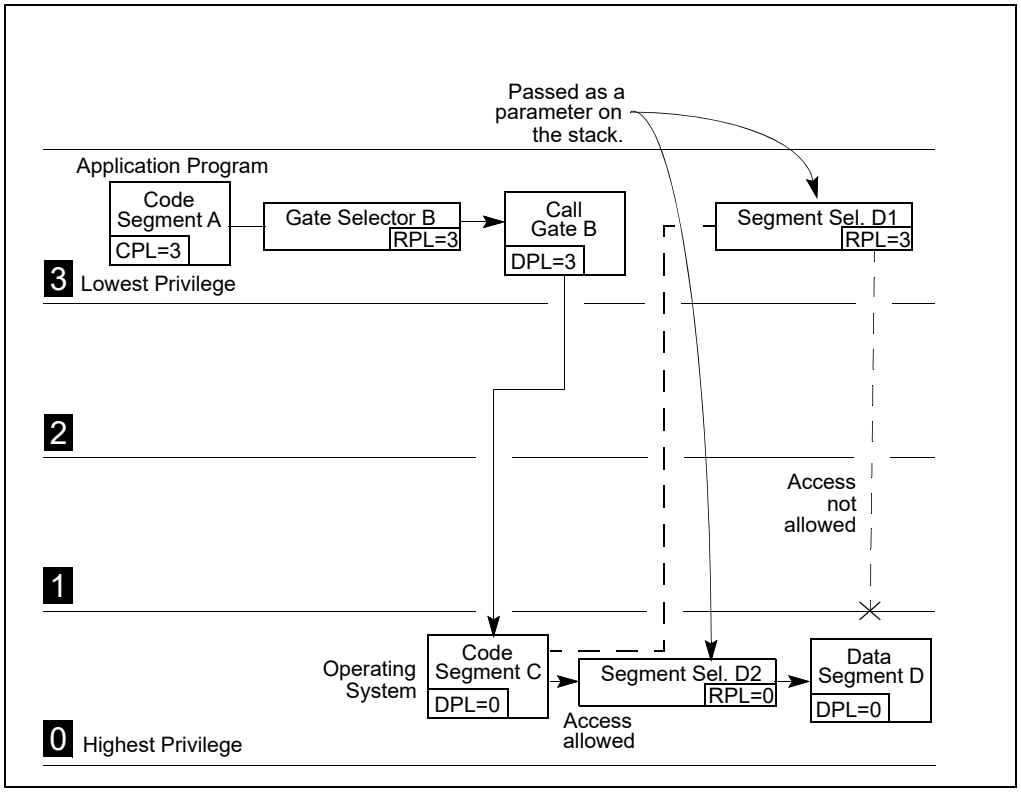


Figure 6-15. Use of RPL to Weaken Privilege Level of Called Procedure

Now assume that instead of setting the RPL of the segment selector to 3, the application program sets the RPL to 0 (segment selector D2). The operating system can now access data segment D, because its CPL and the RPL of segment selector D2 are both equal to the DPL of data segment D.

Because the application program is able to change the RPL of a segment selector to any value, it can potentially use a procedure operating at a numerically lower privilege level to access a protected data structure. This ability to lower the RPL of a segment selector breaches the processor's protection mechanism.

Because a called procedure cannot rely on the calling procedure to set the RPL correctly, operating-system procedures (executing at numerically lower privilege-levels) that receive segment selectors from numerically higher privilege-level procedures need to test the RPL of the segment selector to determine if it is at the appropriate level. The ARPL (adjust requested privilege level) instruction is provided for this purpose. This instruction adjusts the RPL of one segment selector to match that of another segment selector.

The example in Figure 6-15 demonstrates how the ARPL instruction is intended to be used. When the operating-system receives segment selector D2 from the application program, it uses the ARPL instruction to compare the RPL of the segment selector with the privilege level of the application program (represented by the code-segment selector pushed onto the stack). If the RPL is less than application program's privilege level, the ARPL instruction changes the RPL of the segment selector to match the privilege level of the application program (segment selector D1). Using this instruction thus prevents a procedure running at a numerically higher privilege level from accessing numerically lower privilege-level (more privileged) segments by lowering the RPL of a segment selector.

Note that the privilege level of the application program can be determined by reading the RPL field of the segment selector for the application-program's code segment. This segment selector is stored on the stack as part of the call to the operating system. The operating system can copy the segment selector from the stack into a register for use as an operand for the ARPL instruction.

6.10.5 Checking Alignment

When the CPL is 3, alignment of memory references can be checked by setting the AM flag in the CR0 register and the AC flag in the EFLAGS register. Unaligned memory references generate alignment exceptions (#AC). The processor does not generate alignment exceptions when operating at privilege level 0, 1, or 2. See Table 7-7 for a description of the alignment requirements when alignment checking is enabled.

6.11 PAGE-LEVEL PROTECTION

Page-level protection can be used alone or applied to segments. When page-level protection is used with the flat memory model, it allows supervisor code and data (the operating system or executive) to be protected from user code and data (application programs). It also allows pages containing code to be write protected. When the segment- and page-level protection are combined, page-level read/write protection allows more protection granularity within segments.

With page-level protection (as with segment-level protection) each memory reference is checked to verify that protection checks are satisfied. All checks are made before the memory cycle is started, and any violation prevents the cycle from starting and results in a page-fault exception being generated. Because checks are performed in parallel with address translation, there is no performance penalty.

The processor performs two page-level protection checks:

- Restriction of addressable domain (supervisor and user modes).
- Page type (read only or read/write).

Violations of either of these checks results in a page-fault exception being generated. See Chapter 7, "Interrupt 14—Page-Fault Exception (#PF)," for an explanation of the page-fault exception mechanism. This chapter describes the protection violations which lead to page-fault exceptions.

6.11.1 Page-Protection Flags

Protection information for pages is contained in two flags in a paging-structure entry (see Chapter 5): the read/write flag (bit 1) and the user/supervisor flag (bit 2). The protection checks use the flags in all paging structures.

6.11.2 Restricting Addressable Domain

The page-level protection mechanism allows restricting access to pages based on two privilege levels:

- Supervisor mode (U/S flag is 0)—(Most privileged) For the operating system or executive, other system software (such as device drivers), and protected system data (such as page tables).
- User mode (U/S flag is 1)—(Least privileged) For application code and data.

The segment privilege levels map to the page privilege levels as follows. If the processor is currently operating at a CPL of 0, 1, or 2, it is in supervisor mode; if it is operating at a CPL of 3, it is in user mode. When the processor is in supervisor mode, it can access all pages; when in user mode, it can access only user-level pages. (Note that the WP flag in control register CR0 modifies the supervisor permissions, as described in Section 6.11.3, "Page Type.")

Note that to use the page-level protection mechanism, code and data segments must be set up for at least two segment-based privilege levels: level 0 for supervisor code and data segments and level 3 for user code and data segments. (In this model, the stacks are placed in the data segments.) To minimize the use of segments, a flat memory model can be used (see Section 3.2.1, "Basic Flat Model").

Here, the user and supervisor code and data segments all begin at address zero in the linear address space and overlay each other. With this arrangement, operating-system code (running at the supervisor level) and application code (running at the user level) can execute as if there are no segments. Protection between operating-system and application code and data is provided by the processor's page-level protection mechanism.

6.11.3 Page Type

The page-level protection mechanism recognizes two page types:

- Read-only access (R/W flag is 0).
- Read/write access (R/W flag is 1).

When the processor is in supervisor mode and the WP flag in register CR0 is clear (its state following reset initialization), all pages are both readable and writable (write-protection is ignored). When the processor is in user mode, it can write only to user-mode pages that are read/write accessible. User-mode pages which are read/write or read-only are readable; supervisor-mode pages are neither readable nor writable from user mode. A page-fault exception is generated on any attempt to violate the protection rules.

Starting with the P6 family, Intel processors allow user-mode pages to be write-protected against supervisor-mode access. Setting CR0.WP = 1 enables supervisor-mode sensitivity to write protected pages. If CR0.WP = 1, read-only pages are not writable from any privilege level. This supervisor write-protect feature is useful for implementing a "copy-on-write" strategy used by some operating systems, such as UNIX*, for task creation (also called forking or spawning). When a new task is created, it is possible to copy the entire address space of the parent task. This gives the child task a complete, duplicate set of the parent's segments and pages. An alternative copy-on-write strategy saves memory space and time by mapping the child's segments and pages to the same segments and pages used by the parent task. A private copy of a page gets created only when one of the tasks writes to the page. By using the WP flag and marking the shared pages as read-only, the supervisor can detect an attempt to write to a page, and can copy the page at that time.

6.11.4 Combining Protection of Both Levels of Page Tables

For any one page, the protection attributes of its page-directory entry (first-level page table) may differ from those of its page-table entry (second-level page table). The processor checks the protection for a page in both its page-directory and the page-table entries. Table 6-3 shows the protection provided by the possible combinations of protection attributes when the WP flag is clear.

6.11.5 Overrides to Page Protection

The following types of memory accesses are checked as if they are privilege-level 0 accesses, regardless of the CPL at which the processor is currently operating:

- Access to segment descriptors in the GDT, LDT, or IDT.
- Access to an inner-privilege-level stack during an inter-privilege-level call or a call to in exception or interrupt handler, when a change of privilege level occurs.

6.12 COMBINING PAGE AND SEGMENT PROTECTION

When paging is enabled, the processor evaluates segment protection first, then evaluates page protection. If the processor detects a protection violation at either the segment level or the page level, the memory access is not carried out and an exception is generated. If an exception is generated by segmentation, no paging exception is generated.

Page-level protections cannot be used to override segment-level protection. For example, a code segment is by definition not writable. If a code segment is paged, setting the R/W flag for the pages to read-write does not make the pages writable. Attempts to write into the pages will be blocked by segment-level protection checks.

Page-level protection can be used to enhance segment-level protection. For example, if a large read-write data segment is paged, the page-protection mechanism can be used to write-protect individual pages.

Table 6-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	**Access Type**	**Privilege**	**Access Type**	**Privilege**	**Access Type**
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

NOTE:

* If CR0.WP = 1, access type is determined by the R/W flags of the page-directory and page-table entries. IF CR0.WP = 0, supervisor privilege permits read-write access.

6.13 PAGE-LEVEL PROTECTION AND EXECUTE-DISABLE BIT

In addition to page-level protection offered by the U/S and R/W flags, paging structures used with PAE paging, 4-level paging,[1] and 5-level paging provide the execute-disable bit (see Chapter 5, "Paging"). This bit offers additional protection for data pages.

An Intel 64 or IA-32 processor with the execute-disable bit capability can prevent data pages from being used by malicious software to execute code. This capability is provided in:

- 32-bit protected mode with PAE enabled.
- IA-32e mode.

While the execute-disable bit capability does not introduce new instructions, it does require operating systems to use a PAE-enabled environment and establish a page-granular protection policy for memory pages.

If the execute-disable bit of a memory page is set, that page can be used only as data. An attempt to execute code from a memory page with the execute-disable bit set causes a page-fault exception.

The execute-disable capability is not supported with 32-bit paging. Existing page-level protection mechanisms (see Section 6.11, "Page-Level Protection") continue to apply to memory pages independent of the execute-disable setting.

6.13.1 Detecting and Enabling the Execute-Disable Capability

Software can detect the presence of the execute-disable capability using the CPUID instruction. CPUID.80000001H:EDX.NX [bit 20] = 1 indicates the capability is available.

If the capability is available, software can enable it by setting IA32_EFER.NXE[bit 11] to 1. IA32_EFER is available if CPUID.80000001H.EDX[bit 20 or 29] = 1.

If the execute-disable capability is not available, a write to set IA32_EFER.NXE produces a #GP exception. See Table 6-4.

Table 6-4. Extended Feature Enable MSR (IA32_EFER)

63:12	11	10	9	8	7:1	0
Reserved	Execute-disable bit enable (NXE)	IA-32e mode active (LMA)	Reserved	IA-32e mode enable (LME)	Reserved	SysCall enable (SCE)

6.13.2 Execute-Disable Page Protection

The execute-disable bit in the paging structures enhances page protection for data pages. Instructions cannot be fetched from a memory page if IA32_EFER.NXE =1 and the execute-disable bit is set in any of the paging-structure entries used to map the page. Table 6-5 lists the valid usage of a page in relation to the value of execute-disable bit (bit 63) of the corresponding entry in each level of the paging structures. Execute-disable protection can be activated using the execute-disable bit at any level of the paging structure, irrespective of the corresponding entry in other levels. When execute-disable protection is not activated, the page can be used as code or data.

1. Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.

Table 6-5. Page Level Protection Matrix with Execute-Disable Bit Capability with 4-Level Paging

Execute Disable Bit Value (Bit 63)				Valid Usage
PML4	PDP	PDE	PTE	
Bit 63 = 1	*	*	*	Data
*	Bit 63 = 1	*	*	Data
*	*	Bit 63 = 1	*	Data
*	*	*	Bit 63 = 1	Data
Bit 63 = 0	Bit 63 = 0	Bit 63 = 0	Bit 63 = 0	Data/Code

NOTES:

* Value not checked.

In legacy PAE-enabled mode, Table 6-6 and Table 6-7 show the effect of setting the execute-disable bit for code and data pages.

Table 6-6. 4-KByte Page Level Protection Matrix with Execute-Disable Bit Capability with PAE Paging

Execute Disable Bit Value (Bit 63)		Valid Usage
PDE	PTE	
Bit 63 = 1	*	Data
*	Bit 63 = 1	Data
Bit 63 = 0	Bit 63 = 0	Data/Code

NOTE:

* Value not checked.

Table 6-7. 2-MByte Page Level Protection with Execute-Disable Bit Capability with PAE Paging

Execute Disable Bit Value (Bit 63)	Valid Usage
PDE	
Bit 63 = 1	Data
Bit 63 = 0	Data/Code

6.13.3 Reserved Bit Checking

The processor enforces reserved bit checking in paging data structure entries. The bits being checked varies with paging mode and may vary with the size of physical address space.

Table 6-8 shows the reserved bits that are checked when the execute disable bit capability is enabled (CR4.PAE = 1 and IA32_EFER.NXE = 1). Table 6-8 and Table 6-9 show the following paging modes:

- Non-PAE 4-KByte paging: 4-KByte-page only paging (CR4.PAE = 0, CR4.PSE = 0).
- PSE36: 4-KByte and 4-MByte pages (CR4.PAE = 0, CR4.PSE = 1).
- PAE: 4-KByte and 2-MByte pages (CR4.PAE = 1, CR4.PSE = X).

The reserved bit checking depends on the physical address size supported by the implementation, which is reported in CPUID.80000008H. See the table note.

Table 6-8. Page Level Protection Matrix with Execute-Disable Bit Capability Enabled

Mode	Paging Mode	Check Bits
32-bit	4-KByte paging (non-PAE)	No reserved bits checked
	PSE36 - PDE, 4-MByte page	Bit [21]
	PSE36 - PDE, 4-KByte page	No reserved bits checked
	PSE36 - PTE	No reserved bits checked
	PAE - PDP table entry	Bits [63:MAXPHYADDR] & [8:5] & [2:1] *
	PAE - PDE, 2-MByte page	Bits [62:MAXPHYADDR] & [20:13] *
	PAE - PDE, 4-KByte page	Bits [62:MAXPHYADDR] *
	PAE - PTE	Bits [62:MAXPHYADDR] *
64-bit	PML5E	Bits [51:MAXPHYADDR] *
	PML4E	Bits [51:MAXPHYADDR] *
	PDPTE	Bits [51:MAXPHYADDR] *
	PDE, 2-MByte page	Bits [51:MAXPHYADDR] & [20:13] *
	PDE, 4-KByte page	Bits [51:MAXPHYADDR] *
	PTE	Bits [51:MAXPHYADDR] *

NOTES:

* MAXPHYADDR is the maximum physical address size and is indicated by CPUID.80000008H:EAX[bits 7-0].

If execute disable bit capability is not enabled or not available, reserved bit checking in 64-bit mode includes bit 63 and additional bits. This and reserved bit checking for legacy 32-bit paging modes are shown in Table 6-10.

Table 6-9. Reserved Bit Checking with Execute-Disable Bit Capability Not Enabled

Mode	Paging Mode	Check Bits
32-bit	KByte paging (non-PAE)	No reserved bits checked
	PSE36 - PDE, 4-MByte page	Bit [21]
	PSE36 - PDE, 4-KByte page	No reserved bits checked
	PSE36 - PTE	No reserved bits checked
	PAE - PDP table entry	Bits [63:MAXPHYADDR] & [8:5] & [2:1]*
	PAE - PDE, 2-MByte page	Bits [63:MAXPHYADDR] & [20:13]*
	PAE - PDE, 4-KByte page	Bits [63:MAXPHYADDR]*
	PAE - PTE	Bits [63:MAXPHYADDR]*
64-bit	PML5E	Bit [63], bits [51:MAXPHYADDR]*
	PML4E	Bit [63], bits [51:MAXPHYADDR]*
	PDPTE	Bit [63], bits [51:MAXPHYADDR]*
	PDE, 2-MByte page	Bit [63], bits [51:MAXPHYADDR] & [20:13]*
	PDE, 4-KByte page	Bit [63], bits [51:MAXPHYADDR]*
	PTE	Bit [63], bits [51:MAXPHYADDR]*

NOTES:

* MAXPHYADDR is the maximum physical address size and is indicated by CPUID.80000008H:EAX[bits 7-0].

6.13.4 Exception Handling

When execute disable bit capability is enabled (IA32_EFER.NXE = 1), conditions for a page fault to occur include the same conditions that apply to an Intel 64 or IA-32 processor without execute disable bit capability plus the

following new condition: an instruction fetch to a linear address that translates to physical address in a memory page that has the execute-disable bit set.

An Execute Disable Bit page fault can occur at all privilege levels. It can occur on any instruction fetch, including (but not limited to): near branches, far branches, CALL/RET/INT/IRET execution, sequential instruction fetches, and task switches. The execute-disable bit in the page translation mechanism is checked only when:

- IA32_EFER.NXE = 1.
- The instruction translation look-aside buffer (ITLB) is loaded with a page that is not already present in the ITLB.

CHAPTER 7
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 22, "8086 Emulation," describes information specific to interrupt and exception mechanisms in real-address and virtual-8086 mode. Section 7.14, "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.

7.1 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.

7.2 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 7.10, "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 7.3, "Sources of Interrupts").

Table 7-1 shows vector number assignments for architecturally defined exceptions and for the NMI interrupt. This table gives the exception type (see Section 7.5, "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.

7.3 SOURCES OF INTERRUPTS

The processor receives interrupts from two sources:

- External (hardware generated) interrupts.
- Software-generated interrupts.

7.3.1 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 12, "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 7.2, "Exception and Interrupt Vectors"). Asserting the NMI pin signals a non-maskable interrupt (NMI), which is assigned to interrupt vector 2.

Table 7-1. Protected-Mode Exceptions and Interrupts

Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[1]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.

Table 7-1. Protected-Mode Exceptions and Interrupts (Contd.)

Vector	Mnemonic	Description	Type	Error Code	Source
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes	Any data reference in memory.[2]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[3]
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[4]
20	#VE	Virtualization Exception	Fault	No	EPT violations[5]
21	#CP	Control Protection Exception	Fault	Yes	RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump.
22-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

NOTES:

1. 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.

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 33, "System Management Mode."

7.3.2 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 7.8.1, "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.

7.3.3 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.

7.4 SOURCES OF EXCEPTIONS

The processor receives exceptions from three sources:

- Processor-detected program-error exceptions.
- Software-generated exceptions.
- Machine-check exceptions.

7.4.1 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 7.5, "Exception Classifications").

7.4.2 Software-Generated Exceptions

The INTO, INT1, INT3, and BOUND instructions permit exceptions to be generated in software. These instructions allow checks for exception conditions to be performed at points in the instruction stream. For example, INT3 causes a breakpoint exception to be generated.

The INT *n* instruction can be used to emulate exceptions in software; but there is a limitation.[1] If INT *n* provides a vector for one of the architecturally-defined exceptions, the processor generates an interrupt to the correct vector (to access the exception handler) but does not push an error code on the stack. This is true even if the associated hardware-generated exception normally produces an error code. The exception handler will still attempt to pop an error code from the stack while handling the exception. Because no error code was pushed, the handler will pop off and discard the EIP instead (in place of the missing error code). This sends the return to the wrong location.

7.4.3 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.

---

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.

See Chapter 7, "Interrupt 18—Machine-Check Exception (#MC)," and Chapter 17, "Machine-Check Architecture," for more information about the machine-check mechanism.

7.5 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.

7.6 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 ensure 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).

7.7 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 ensure that no other interrupts, including NMI interrupts, are received until the NMI handler has completed executing (see Section 7.7.1, "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.

7.7.1 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 7.8.1, "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 22.2.7, "Sensitive Instructions"). In such a case, NMIs are unmasked before the exception handler is invoked.

7.8 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.

7.8.1 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 7.3.2, "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 7.4.2, "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.[1] If IF = 0, maskable hardware interrupts remain inhibited on the instruction boundary following an execution of STI.[2] 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.

7.8.2 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 19.3.1.1, "Instruction-Breakpoint Exception Condition," for more information on the use of this flag.

As noted in Section 7.8.3, 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).

---

1. 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 22.3, "Interrupt and Exception Handling in Virtual-8086 Mode." Behavior is also impacted by the PVI flag: see Section 22.4, "Protected-Mode Virtual Interrupts."
2. Nonmaskable interrupts and system-management interrupts may also be inhibited on the instruction boundary following such an execution of STI.

7.8.3 Masking Exceptions and Interrupts When Switching Stacks

To switch to a different stack segment, software often uses a pair of instructions, for example:

```
MOV SS, AX
MOV ESP, StackTop
```

(Software might also use the POP instruction to load SS and ESP.)

If an interrupt or exception occurs after the new SS segment descriptor has been loaded but before the ESP register has been loaded, these two parts of the logical address into the stack space are inconsistent for the duration of the interrupt or exception handler (assuming that delivery of the interrupt or exception does not itself load a new stack pointer).

To account for this situation, the processor prevents certain events from being delivered after execution of a MOV to SS instruction or a POP to SS instruction. The following items provide details:

- Any instruction breakpoint on the next instruction is suppressed (as if EFLAGS.RF were 1).
- Any data breakpoint on the MOV to SS instruction or POP to SS instruction is inhibited until the instruction boundary following the next instruction.
- Any single-step trap that would be delivered following the MOV to SS instruction or POP to SS instruction (because EFLAGS.TF is 1) is suppressed.
- The suppression and inhibition ends after delivery of an exception or the execution of the next instruction.
- If a sequence of consecutive instructions each loads the SS register (using MOV or POP), only the first is guaranteed to inhibit or suppress events in this way.

Intel recommends that software use the LSS instruction to load the SS register and ESP together. The problem identified earlier does not apply to LSS, and the LSS instruction does not inhibit events as detailed above.

7.9 PRIORITIZATION OF CONCURRENT EVENTS

If more than one event is pending at an instruction boundary (between execution of instructions), the processor services them in a predictable order. Table 7-2 shows the priority among classes of event sources.

Table 7-2. Priority Among Concurrent Events

Priority	Description
1 (Highest)	Hardware Reset and Machine Checks - RESET - Machine Check (#MC)
2	Trap on Task Switch - T flag in TSS is set (#DB)
3	External Hardware Interventions - FLUSH - STOPCLK - SMI - INIT
4	Traps on the Previous Instruction - Trap-class Debug Exceptions (#DB due to TF flag set or data/I-O breakpoint)
5	Nonmaskable Interrupts (NMI) [1]
6	Maskable Hardware Interrupts [1]
7	Fault-class Debug Exceptions (#DB due to instruction breakpoint)

Table 7-2. Priority Among Concurrent Events (Contd.)

Priority	Description
8	Faults from Fetching Next Instruction - Code-Segment Limit Violation (#GP) - Code Page Fault (#PF)
9 (Lowest)	Faults from Decoding the Next Instruction - Control protection exception due to missing ENDBRANCH at target of an indirect call or jump (#CP) - Instruction length > 15 bytes (#GP) - Invalid Opcode (#UD) - Coprocessor Not Available (#NM)

NOTE

1. The Intel® 486 processor and earlier processors group nonmaskable and maskable interrupts in the same priority class.

The processor first services a pending event 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 may be re-generated when the event handler returns execution to the point in the program or task where the original event occurred. While the priority among the classes listed in Table 7-2 is consistent across processor implementations, the priority of events within a class is implementation-dependent and may vary from processor to processor.

Table 7-2 specifies the prioritization of events that may be pending at an instruction boundary. It does not specify the prioritization of faults that arise during instruction execution or event delivery (these include #BR, #TS, #NP, #SS, #GP, #PF, #AC, #MF, #XM, #VE, or #CP). It also does not apply to the events generated by the "Call to Interrupt Procedure" instructions (INT n, INTO, INT3, and INT1), as these events are integral to the execution of those instructions and do not occur between instructions.

7.10 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 7-1, 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.

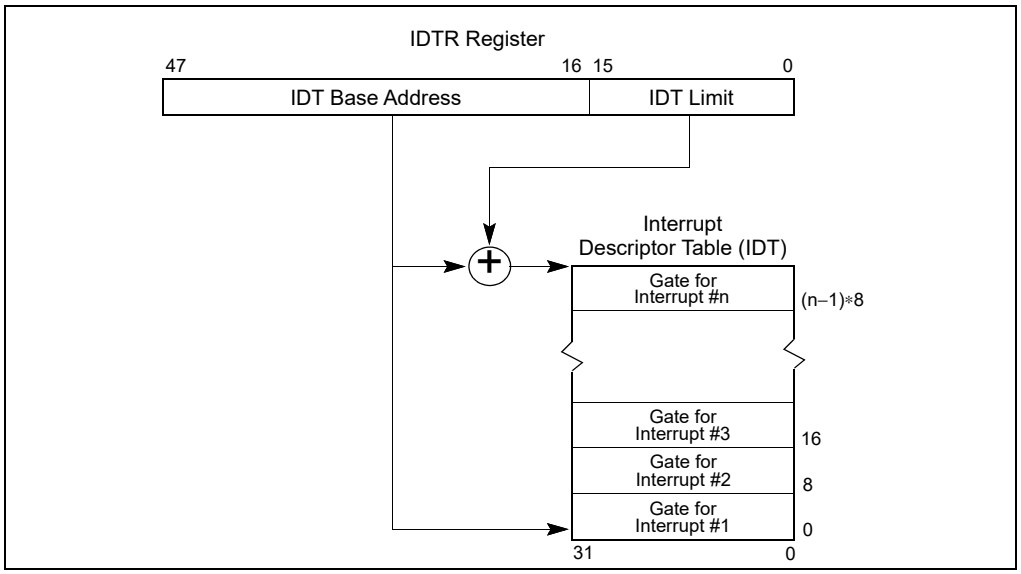


Figure 7-1. Relationship of the IDTR and IDT

7.11 IDT DESCRIPTORS

The IDT may contain any of three kinds of gate descriptors:

- Task-gate descriptor
- Interrupt-gate descriptor
- Trap-gate descriptor

Figure 7-2 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 9.2.5, "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 6.8.3, "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 7.12.1.3, "Flag Usage By Exception- or Interrupt-Handler Procedure").

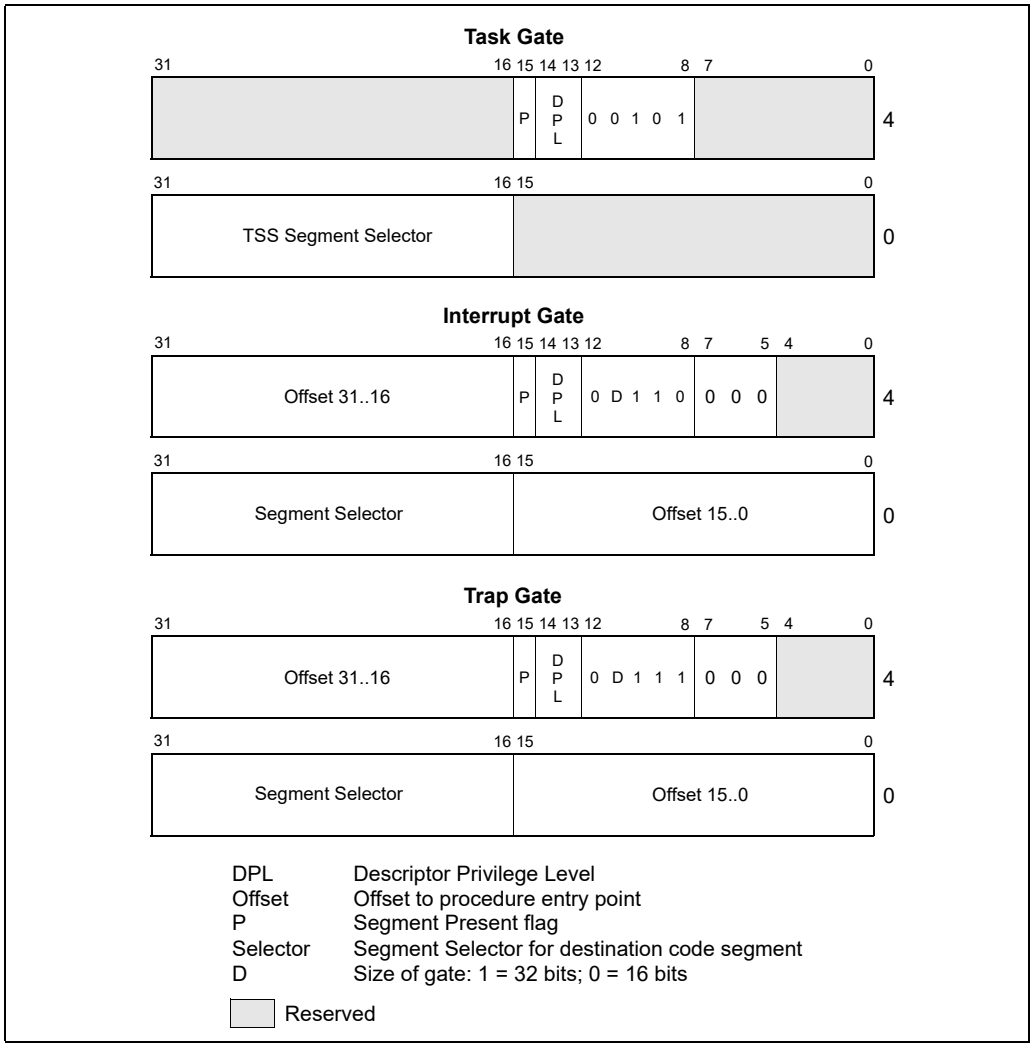


Figure 7-2. IDT Gate Descriptors

7.12 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 6.8.2, “Gate Descriptors,” through Section 6.8.6, “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 9.3, “Task Switching”).

7.12.1 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 7-3). 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.

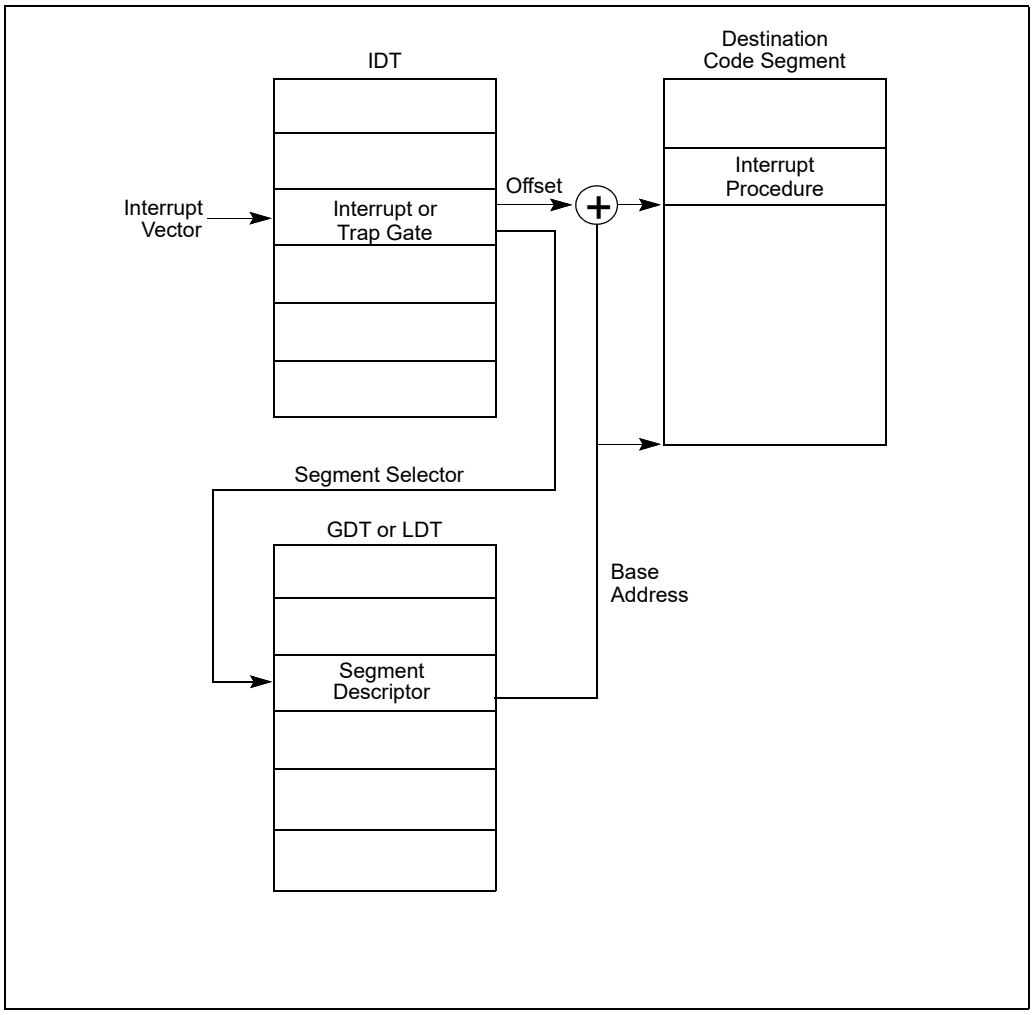


Figure 7-3. Interrupt Procedure Call

When the processor performs a call to the exception- or interrupt-handler procedure:

- If the handler procedure is going to be executed at a numerically lower privilege level, a stack switch occurs. When the stack switch occurs:
  a. The segment selector and stack pointer for the stack to be used by the handler are obtained from the TSS for the currently executing task. On this new stack, the processor pushes the stack segment selector and stack pointer of the interrupted procedure.
  b. The processor then saves the current state of the EFLAGS, CS, and EIP registers on the new stack (see Figure 7-4).
  c. If an exception causes an error code to be saved, it is pushed on the new stack after the EIP value.
- If the handler procedure is going to be executed at the same privilege level as the interrupted procedure:
  a. The processor saves the current state of the EFLAGS, CS, and EIP registers on the current stack (see Figure 7-4).
  b. If an exception causes an error code to be saved, it is pushed on the current stack after the EIP value.

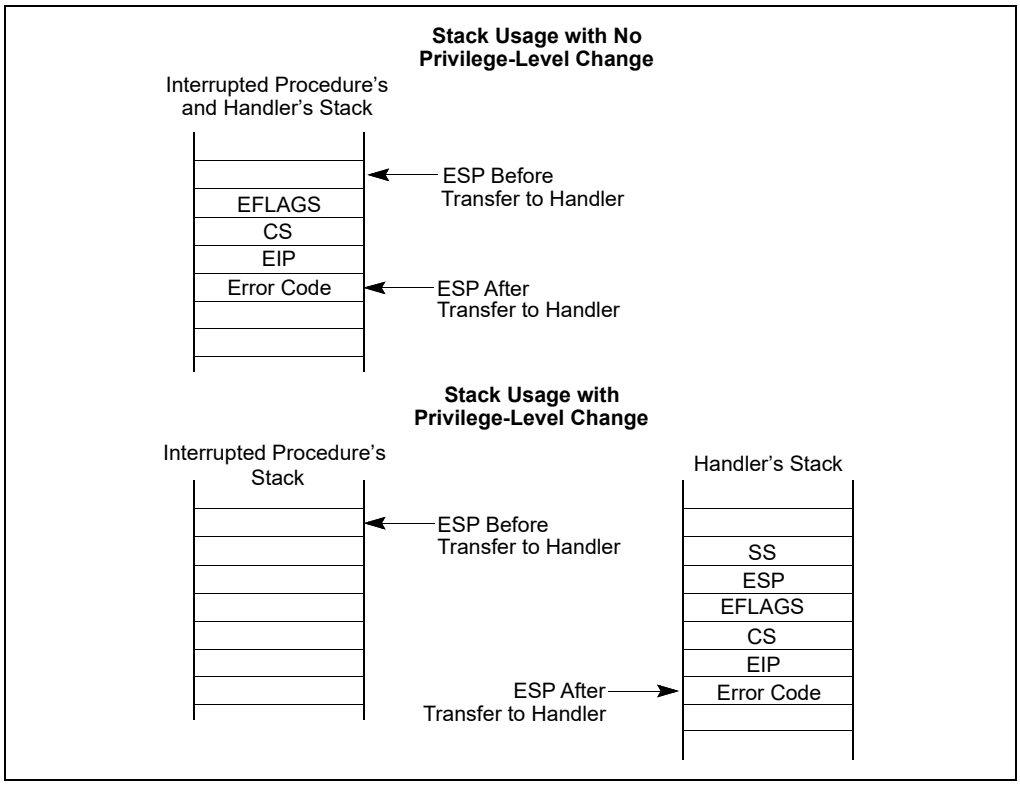


Figure 7-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

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.

7.12.1.1 Shadow Stack Usage on Transfers to Interrupt and Exception Handling Routines

When the processor performs a call to the exception- or interrupt-handler procedure:

- If the handler procedure is going to be execute at a numerically lower privilege level, a shadow stack switch occurs. When the shadow stack switch occurs:
  a. On a transfer from privilege level 3, if shadow stacks are enabled at privilege level 3 then the SSP is saved to the IA32_PL3_SSP MSR.
  b. If shadow stacks are enabled at the privilege level where the handler will execute then the shadow stack for the handler is obtained from one of the following MSRs based on the privilege level at which the handler executes.
     - IA32_PL2_SSP if handler executes at privilege level 2.
     - IA32_PL1_SSP if handler executes at privilege level 1.
     - IA32_PL0_SSP if handler executes at privilege level 0.
  c. The SSP obtained is then verified to ensure it points to a valid supervisory shadow stack that is not currently active by verifying a supervisor shadow stack token at the address pointed to by the SSP. The operations performed to verify and acquire the supervisor shadow stack token by making it busy are as described in Section 18.2.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.
  d. On this new shadow stack, the processor pushes the CS, LIP (CS.base + EIP), and SSP of the interrupted procedure if the interrupted procedure was executing at privilege level less than 3; see Figure 7-5.[1]
- If the handler procedure is going to be executed at the same privilege level as the interrupted procedure and shadow stacks are enabled at current privilege level:
  a. The processor saves the current state of the CS, LIP (CS.base + EIP), and SSP registers on the current shadow stack; see Figure 7-5.

1. If any of these pushes leads to an exception or a VM exit, the supervisor shadow-stack token remains busy.

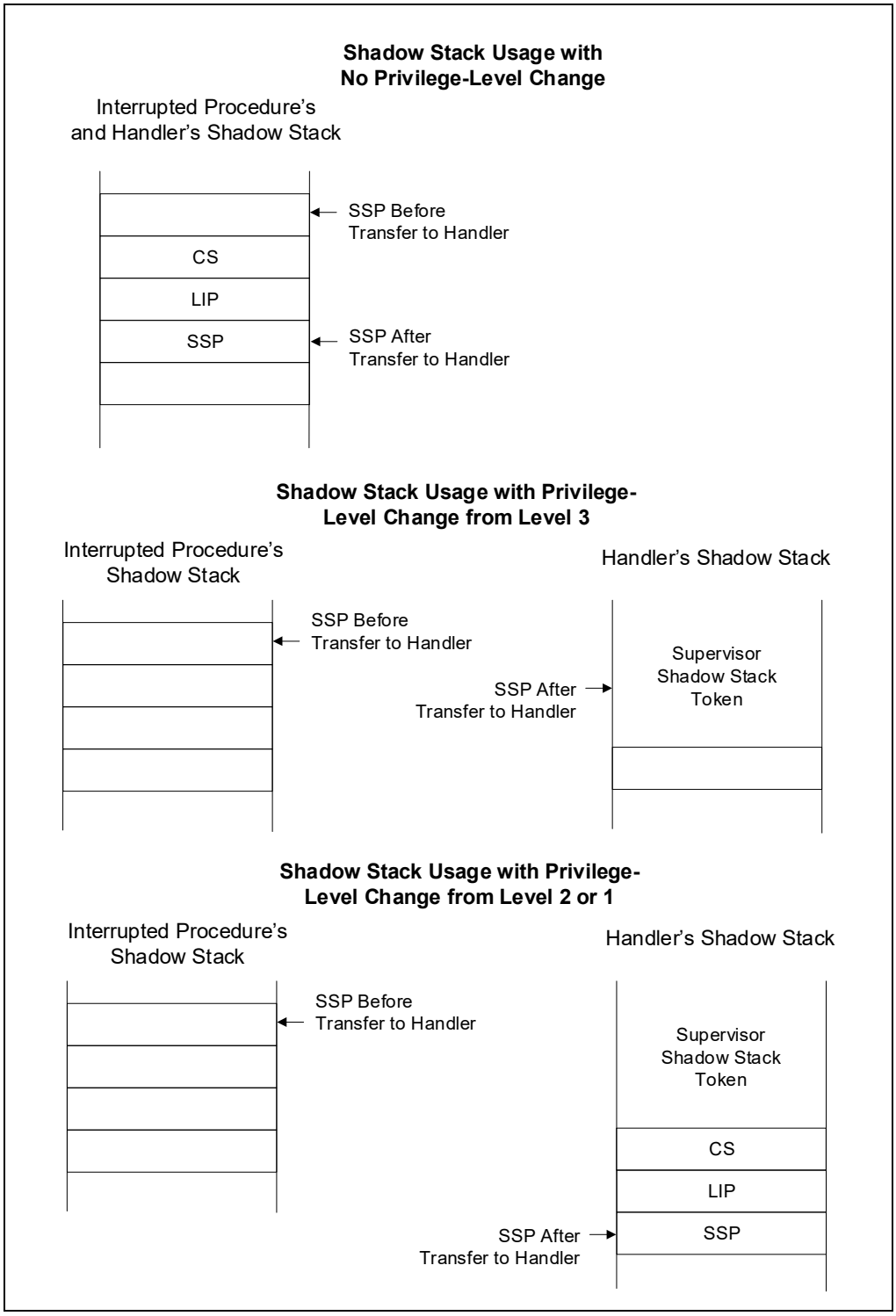


Figure 7-5. Shadow Stack Usage on Transfers to Interrupt and Exception-Handling Routines

To return from an exception- or interrupt-handler procedure, the handler must use the IRET (or IRETD) instruction.

When executing a return from an interrupt or exception handler from the same privilege level as the interrupted procedure, the processor performs these actions to enforce return address protection:

- Restores the CS and EIP registers to their values prior to the interrupt or exception.

If shadow stack is enabled:

— Compares the values on shadow stack at address SSP+8 (the LIP) and SSP+16 (the CS) to the CS and (CS.base + EIP) popped from the stack and causes a control protection exception (#CP(FAR-RET/IRET)) if they do not match.

— Pops the top-of-stack value (the SSP prior to the interrupt or exception) from shadow stack into SSP register.

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

- If shadow stack is enabled at current privilege level:
  - — If SSP is not aligned to 8 bytes then causes a control protection exception (#CP(FAR-RET/IRET)).
  - — If privilege level of the procedure being returned to is less than 3 (returning to supervisor mode):
    - Compares the values on shadow stack at address SSP+8 (the LIP) and SSP+16 (the CS) to the CS and (CS.base + EIP) popped from the stack and causes a control protection exception (#CP(FAR-RET/IRET)) if they do not match.
    - Temporarily saves the top-of-stack value (the SSP of the procedure being returned to) internally.
  - — If a busy supervisor shadow stack token is present at address SSP+24, then marks the token free using operations described in section Section 18.2.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.
  - — If the privilege level of the procedure being returned to is less than 3 (returning to supervisor mode), restores the SSP register from the internally saved value.
  - — If the privilege level of the procedure being returned to is 3 (returning to user mode) and shadow stack is enabled at privilege level 3, then restores the SSP register with value of IA32_PL3_SSP MSR.

7.12.1.2 Protection of Exception- and Interrupt-Handler Procedures

The privilege-level protection for exception- and interrupt-handler procedures is similar to that used for ordinary procedure calls when called through a call gate (see Section 6.8.4, "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.[1] Here, the CPL must be less than or equal to the DPL of the gate. This restriction prevents application programs or procedures running at privilege level 3 from using a software interrupt to access critical exception handlers, such as the page-fault handler, providing that those handlers are placed in more privileged code segments (numerically lower privilege level). For hardware-generated interrupts and processor-detected exceptions, the processor ignores the DPL of interrupt and trap gates.

Because exceptions and interrupts generally do not occur at predictable times, these privilege rules effectively impose restrictions on the privilege levels at which exception and interrupt- handling procedures can run. Either of the following techniques can be used to avoid privilege-level violations.

- The exception or interrupt handler can be placed in a conforming code segment. This technique can be used for handlers that only need to access data available on the stack (for example, divide error exceptions). If the handler needs data from a data segment, the data segment needs to be accessible from privilege level 3, which would make it unprotected.
- The handler can be placed in a nonconforming code segment with privilege level 0. This handler would always run, regardless of the CPL that the interrupted program or task is running at.

1. 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).

7.12.1.3 Flag Usage By Exception- or Interrupt-Handler Procedure

When accessing an exception or interrupt handler through either an interrupt gate or a trap gate, the processor clears the TF flag in the EFLAGS register after it saves the contents of the EFLAGS register on the stack. (On calls to exception and interrupt handlers, the processor also clears the VM, RF, and NT flags in the EFLAGS register, after they are saved on the stack.) Clearing the TF flag prevents instruction tracing from affecting interrupt response and ensures that no single-step exception will be delivered after delivery to the handler. A subsequent IRET instruction restores the TF (and VM, RF, and NT) flags to the values in the saved contents of the EFLAGS register on the stack.

The only difference between an interrupt gate and a trap gate is the way the processor handles the IF flag in the EFLAGS register. When accessing an exception- or interrupt-handling procedure through an interrupt gate, the processor clears the IF flag to prevent other interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores the IF flag to its value in the saved contents of the EFLAGS register on the stack. Accessing a handler procedure through a trap gate does not affect the IF flag.

7.12.2 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 7-6). A switch to the handler task is handled in the same manner as an ordinary task switch (see Section 9.3, “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.

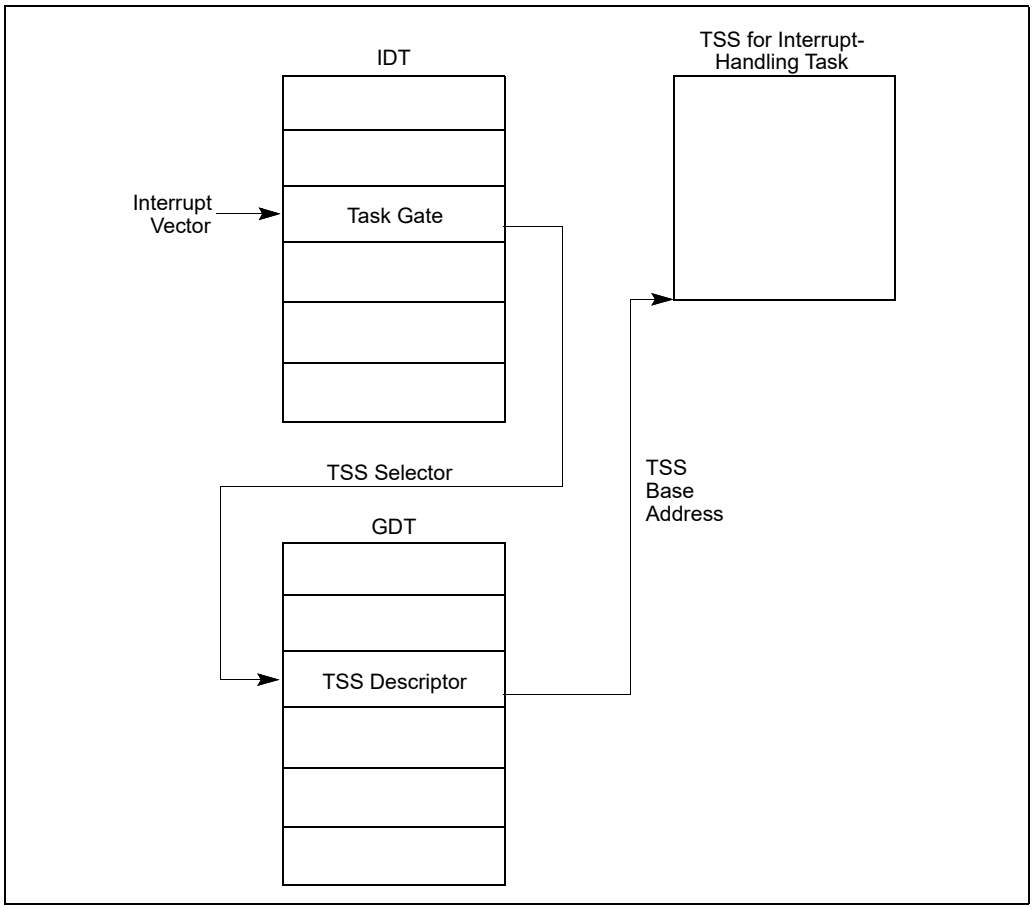


Figure 7-6. Interrupt Task Switch

7.13 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 7-7. The error code resembles a segment selector; however, instead of a TI flag and RPL field, the error code contains 3 flags:

EXT — **External event (bit 0)** — When set, indicates that the exception occurred during delivery of an event external to the program, such as an interrupt or an earlier exception.[1] The bit is cleared if the exception occurred during delivery of a software interrupt (INT *n*, INT3, or INTO).

IDT — **Descriptor location (bit 1)** — When set, indicates that the index portion of the error code refers to a gate descriptor in the IDT; when clear, indicates that the index refers to a descriptor in the GDT or the current LDT.

TI — **GDT/LDT (bit 2)** — Only used when the IDT flag is clear. When set, the TI flag indicates that the index portion of the error code refers to a segment or gate descriptor in the LDT; when clear, it indicates that the index refers to a descriptor in the current GDT.

1. The bit is also set if the exception occurred during delivery of INT1.

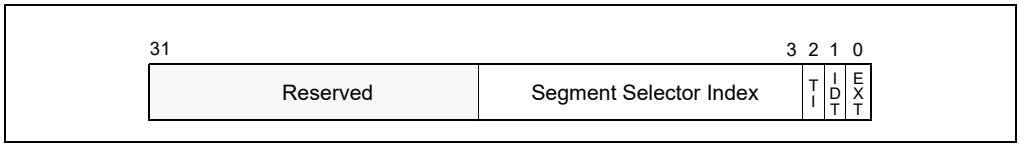


Figure 7-7. Error Code

The segment selector index field provides an index into the IDT, GDT, or current LDT to the segment or gate selector being referenced by the error code. In some cases the error code is null (all bits are clear except possibly EXT). A null error code indicates that the error was not caused by a reference to a specific segment or that a null segment selector was referenced in an operation.

The format of the error code is different for page-fault exceptions (#PF). See the "Interrupt 14—Page-Fault Exception (#PF)" section in this chapter.

The format of the error code is different for control protection exceptions (#CP). See the "Interrupt 21—Control Protection Exception (#CP)" section in this chapter.

The error code is pushed on the stack as a doubleword or word (depending on the default interrupt, trap, or task gate size). To keep the stack aligned for doubleword pushes, the upper half of the error code is reserved. Note that the error code is not popped when the IRET instruction is executed to return from an exception handler, so the handler must remove the error code before executing a return.

Error codes are not pushed on the stack for exceptions that are generated externally (with the INTR or LINT[1:0] pins) or the INT *n* instruction, even if an error code is normally produced for those exceptions.

7.14 EXCEPTION AND INTERRUPT HANDLING IN 64-BIT MODE

In 64-bit mode, interrupt and exception handling is similar to what has been described for non-64-bit modes. The following are the exceptions:

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

7.14.1 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 7-8.

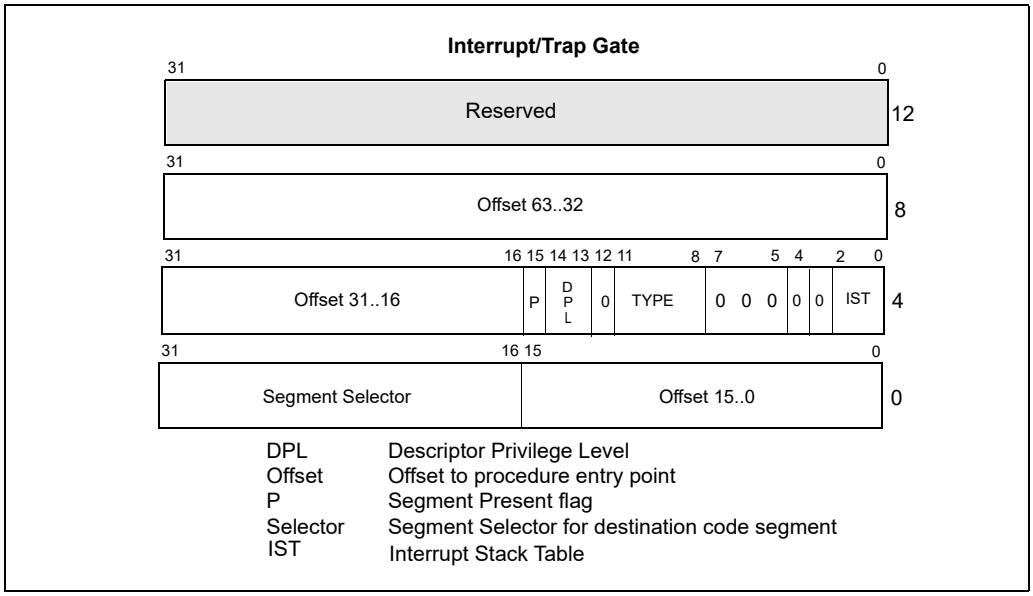


Figure 7-8. 64-Bit IDT Gate Descriptors

In 64-bit mode, the IDT index is formed by scaling the interrupt vector by 16. The first eight bytes (bytes 7:0) of a 64-bit mode interrupt gate are similar but not identical to legacy 32-bit interrupt gates. The type field (bits 11:8 in bytes 7:4) is described in Table 3-2. The Interrupt Stack Table (IST) field (bits 4:0 in bytes 7:4) is used by the stack switching mechanisms described in Section 7.14.5, "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.

7.14.2 64-Bit Mode Stack Frame

In legacy mode, the size of an IDT entry (16 bits or 32 bits) determines the size of interrupt-stack-frame pushes. SS:ESP is pushed only on a CPL change. In 64-bit mode, the size of interrupt stack-frame pushes is fixed at eight bytes. This is because only 64-bit mode gates can be referenced. 64-bit mode also pushes SS:RSP unconditionally, rather than only on a CPL change.

When shadow stacks are enabled at the interrupt handler's privilege level and the interrupted procedure was not executing at a privilege level 3, then the processor pushes the CS:LIP:SSP of the interrupted procedure on the shadow stack of the interrupt handler (where LIP is the linear address of the return address).

Aside from error codes, pushing SS:RSP unconditionally presents operating systems with a consistent interrupt-stackframe size across all interrupts. Interrupt service-routine entry points that handle interrupts generated by the INTn instruction or external INTR# signal can push an additional error code place-holder to maintain consistency.

In legacy mode, the stack pointer may be at any alignment when an interrupt or exception causes a stack frame to be pushed. This causes the stack frame and succeeding pushes done by an interrupt handler to be at arbitrary alignments. In IA-32e mode, the RSP is aligned to a 16-byte boundary before pushing the stack frame. The stack frame itself is aligned on a 16-byte boundary when the interrupt handler is called. The processor can arbitrarily realign the new RSP on interrupts because the previous (possibly unaligned) RSP is unconditionally saved on the newly aligned stack. The previous RSP will be automatically restored by a subsequent IRET.

Aligning the stack permits exception and interrupt frames to be aligned on a 16-byte boundary before interrupts are re-enabled. This allows the stack to be formatted for optimal storage of 16-byte XMM registers, which enables the interrupt handler to use faster 16-byte aligned loads and stores (MOVAPS rather than MOVUPS) to save and restore XMM registers.

Although the RSP alignment is always performed when LMA = 1, it is only of consequence for the kernel-mode case where there is no stack switch or IST used. For a stack switch or IST, the OS would have presumably put suitably aligned RSP values in the TSS.

7.14.3 IRET in IA-32e Mode

In IA-32e mode, IRET executes with an 8-byte operand size. There is nothing that forces this requirement. The stack is formatted in such a way that for actions where IRET is required, the 8-byte IRET operand size works correctly.

Because interrupt stack-frame pushes are always eight bytes in IA-32e mode, an IRET must pop eight byte items off the stack. This is accomplished by preceding the IRET with a 64-bit operand-size prefix. The size of the pop is determined by the address size of the instruction. The SS/ESP/RSP size adjustment is determined by the stack size.

IRET pops SS:RSP unconditionally off the interrupt stack frame only when it is executed in 64-bit mode. In compatibility mode, IRET pops SS:RSP off the stack only if there is a CPL change. This allows legacy applications to execute properly in compatibility mode when using the IRET instruction. 64-bit interrupt service routines that exit with an IRET unconditionally pop SS:RSP off of the interrupt stack frame, even if the target code segment is running in 64-bit mode or at CPL = 0. This is because the original interrupt always pushes SS:RSP.

When shadow stacks are enabled and the target privilege level is not 3, the CS:LIP from the shadow stack frame is compared to the return linear address formed by CS:EIP from the stack. If they do not match then the processor caused a control protection exception (#CP(FAR-RET/IRET)), else the processor pops the SSP of the interrupted procedure from the shadow stack. If the target privilege level is 3 and shadow stacks are enabled at privilege level 3, then the SSP for the interrupted procedure is restored from the IA32_PL3_SSP MSR.

In IA-32e mode, IRET is allowed to load a NULL SS under certain conditions. If the target mode is 64-bit mode and the target CPL ? 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.

7.14.4 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 7-9). On the subsequent IRET, the old SS is popped from the stack and loaded into the SS register.

In summary, a stack switch in IA-32e mode works like the legacy stack switch, except that a new SS selector is not loaded from the TSS. Instead, the new SS is forced to NULL.

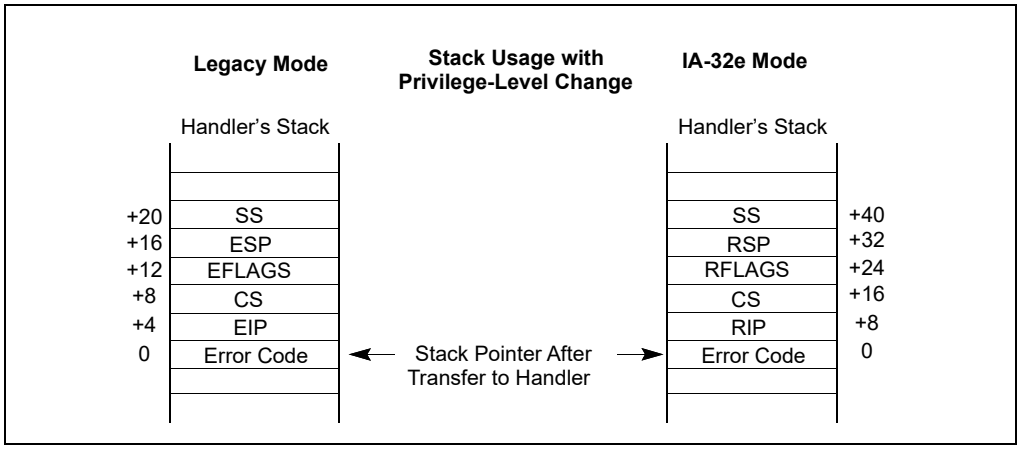


Figure 7-9. IA-32e Mode Stack Usage After Privilege Level Change

7.14.5 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 7-8. The gate descriptor contains a 3-bit IST index field that provides an offset into the IST section of the TSS. Using the IST mechanism, the processor loads the value pointed by an IST pointer into the RSP.

When an interrupt occurs, the new SS selector is forced to NULL and the SS selector's RPL field is set to the new CPL. The old SS, RSP, RFLAGS, CS, and RIP are pushed onto the new stack. Interrupt processing then proceeds as normal. If the IST index is zero, the modified legacy stack-switching mechanism described above is used.

To support this stack-switching mechanism with shadow stacks enabled, the processor provides an MSR, IA32_INTERRUPT_SSP_TABLE, to program the linear address of a table of seven shadow stack pointers that are selected using the IST index from the gate descriptor. To switch to a shadow stack selected from the interrupt shadow stack table pointed to by the IA32_INTERRUPT_SSP_TABLE, the processor requires that the shadow stack addresses programmed into this table point to a supervisor shadow stack token; see Figure 7-10.

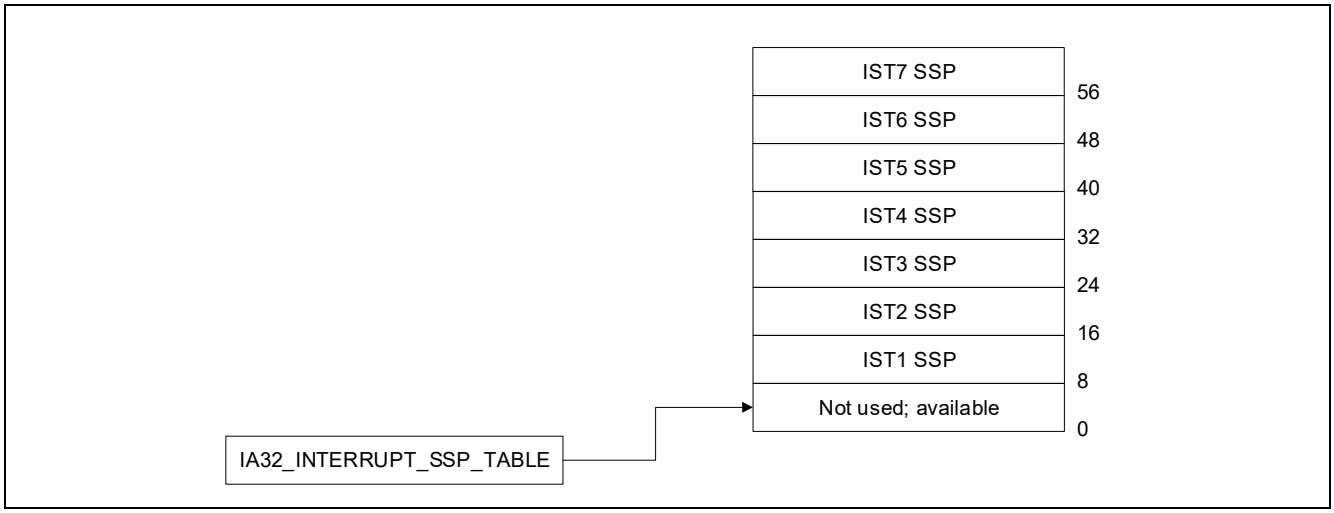


Figure 7-10. Interrupt Shadow Stack Table

7.15 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 7-3). See Chapter 19, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features," for detailed information about the debug exceptions.

Table 7-3. Debug Exception Conditions and Corresponding Exception Classes

Exception Condition	Exception Class
Instruction fetch breakpoint	Fault
Data read or write breakpoint	Trap
I/O read or write breakpoint	Trap
General detect condition (in conjunction with in-circuit emulation)	Fault
Single-step	Trap
Task-switch	Trap
Execution of INT1[1]	Trap

NOTES:

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

Exception Error Code

None. An exception handler can examine the debug registers to determine which condition caused the exception.

Saved Instruction Pointer

Fault — Saved contents of CS and EIP registers point to the instruction that generated the exception.

Trap — Saved contents of CS and EIP registers point to the instruction following the instruction that generated the exception.

Program State Change

Fault — A program-state change does not accompany the debug exception, because the exception occurs before the faulting instruction is executed. The program can resume normal execution upon returning from the debug exception handler.

Trap — A program-state change does accompany the debug exception, because the instruction or task switch being executed is allowed to complete before the exception is generated. However, the new state of the program is not corrupted and execution of the program can continue reliably.

The following items detail the treatment of debug exceptions on the instruction boundary following execution of the MOV or the POP instruction that loads the SS register:

- If EFLAGS.TF is 1, no single-step trap is generated.
- If the instruction encounters a data breakpoint, the resulting debug exception is delivered after completion of the instruction after the MOV or POP. This occurs even if the next instruction is INT *n*, INT3, or INTO.
- Any instruction breakpoint on the instruction after the MOV or POP is suppressed (as if EFLAGS.RF were 1).

Any debug exception inside an RTM region causes a transactional abort and, by default, redirects control flow to the fallback instruction address. If advanced debugging of RTM transactional regions has been enabled, any transactional abort due to a debug exception instead causes execution to roll back to just before the XBEGIN instruction

and then delivers a #DB. See Section 17.3.7, "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 7.5, "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 19.3.2, "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 17.3.7, "RTM-Enabled Debugger Support," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

A breakpoint exception can also be generated by executing the INT *n* instruction with an operand of 3. The action of this instruction (INT 3) is slightly different than that of the INT3 instruction (see "INT n/INTO/INT3/INT1—Call to Interrupt Procedure" in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A).

Exception Error Code

None.

Saved Instruction Pointer

Saved contents of CS and EIP registers point to the instruction following the INT3 instruction.

Program State Change

Even though the EIP points to the instruction following the breakpoint instruction, the state of the program is essentially unchanged because the INT3 instruction does not affect any register or memory locations. The debugger can thus resume the suspended program by replacing the INT3 instruction that caused the breakpoint with the original opcode and decrementing the saved contents of the EIP register. Upon returning from the debugger, program execution resumes with the replaced instruction.

Interrupt 4—Overflow Exception (#OF)

Exception Class Trap.

Description

Indicates that an overflow trap occurred when an INTO instruction was executed. The INTO instruction checks the state of the OF flag in the EFLAGS register. If the OF flag is set, an overflow trap is generated.

Some arithmetic instructions (such as the ADD and SUB) perform both signed and unsigned arithmetic. These instructions set the OF and CF flags in the EFLAGS register to indicate signed overflow and unsigned overflow, respectively. When performing arithmetic on signed operands, the OF flag can be tested directly or the INTO instruction can be used. The benefit of using the INTO instruction is that if the overflow exception is detected, an exception handler can be called automatically to handle the overflow condition.

Exception Error Code

None.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction following the INTO instruction.

Program State Change

Even though the EIP points to the instruction following the INTO instruction, the state of the program is essentially unchanged because the INTO instruction does not affect any register or memory locations. The program can thus resume normal execution upon returning from the overflow exception handler.

Interrupt 5—BOUND Range Exceeded Exception (#BR)

Exception Class Fault.

Description

Indicates that a BOUND-range-exceeded fault occurred when a BOUND instruction was executed. The BOUND instruction checks that a signed array index is within the upper and lower bounds of an array located in memory. If the array index is not within the bounds of the array, a BOUND-range-exceeded fault is generated.

Exception Error Code

None.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the BOUND instruction that generated the exception.

Program State Change

A program-state change does not accompany the bounds-check fault, because the operands for the BOUND instruction are not modified. Returning from the BOUND-range-exceeded exception handler causes the BOUND instruction to be restarted.

Interrupt 6—Invalid Opcode Exception (#UD)

Exception Class Fault.

Description

Indicates that the processor did one of the following things:

- Attempted to execute an invalid or reserved opcode.
- Attempted to execute an instruction with an operand type that is invalid for its accompanying opcode; for example, the source operand for a LES instruction is not a memory location.
- Attempted to execute an MMX or SSE/SSE2/SSE3 instruction on an Intel 64 or IA-32 processor that does not support the MMX technology or SSE/SSE2/SSE3/SSSE3 extensions, respectively. CPUID feature flags MMX (bit 23), SSE (bit 25), SSE2 (bit 26), SSE3 (ECX, bit 0), SSSE3 (ECX, bit 9) indicate support for these extensions.
- Attempted to execute an MMX instruction or SSE/SSE2/SSE3/SSSE3 SIMD instruction (with the exception of the MOVNTI, PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, CLFLUSH, MONITOR, and MWAIT instructions) when the EM flag in control register CR0 is set (1).
- Attempted to execute an SSE/SE2/SSE3/SSSE3 instruction when the OSFXSR bit in control register CR4 is clear (0). Note this does not include the following SSE/SSE2/SSE3 instructions: MASKMOVQ, MOVNTQ, MOVNTI, PREFETCH*h*, SFENCE, LFENCE, MFENCE, and CLFLUSH; or the 64-bit versions of the PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMSKB, PMULHUW, PSADBW, PSHUFW, PADDQ, PSUBQ, PALIGNR, PABSB, PABSD, PABSW, PHADDD, PHADDSW, PHADDW, PHSUBD, PHSUBSW, PHSUBW, PMADDUBSM, PMULHRSW, PSHUFB, PSIGNB, PSIGND, and PSIGNW.
- Attempted to execute an SSE/SSE2/SSE3/SSSE3 instruction on an Intel 64 or IA-32 processor that caused a SIMD floating-point exception when the OSXMMEXCPT bit in control register CR4 is clear (0).
- Executed a UD0, UD1 or UD2 instruction. Note that even though it is the execution of the UD0, UD1 or UD2 instruction that causes the invalid opcode exception, the saved instruction pointer will still points at the UD0, UD1 or UD2 instruction.
- Detected a LOCK prefix that precedes an instruction that may not be locked or one that may be locked but the destination operand is not a memory location.
- Attempted to execute an LLDT, SLDT, LTR, STR, LSL, LAR, VERR, VERW, or ARPL instruction while in real-address or virtual-8086 mode.
- Attempted to execute the RSM instruction when not in SMM mode.

In Intel 64 and IA-32 processors that implement out-of-order execution microarchitectures, this exception is not generated until an attempt is made to retire the result of executing an invalid instruction; that is, decoding and speculatively attempting to execute an invalid opcode does not generate this exception. Likewise, in the Pentium processor and earlier IA-32 processors, this exception is not generated as the result of prefetching and preliminary decoding of an invalid instruction. (See Section 7.5, "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 7-4).

Table 7-4. Interrupt and Exception Classes

Class	Vector Number	Description
Benign Exceptions and Interrupts	1 2 3 4 5 6 7 9 16 17 18 19 All All	Debug NMI Interrupt Breakpoint Overflow BOUND Range Exceeded Invalid Opcode Device Not Available Coprocessor Segment Overrun Floating-Point Error Alignment Check Machine Check SIMD floating-point INT *n* INTR
Contributory Exceptions	0 10 11 12 13 21	Divide Error Invalid TSS Segment Not Present Stack Fault General Protection Control Protection
Page Faults	14 20	Page Fault Virtualization Exception

Table 7-5 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 7-5. 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 7-5. 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 an 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 7-6 shows the conditions that cause an invalid TSS exception to be generated.

Table 7-6. 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 7-6. 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 generated either in the context of the original task or in the context of the new task (see Section 9.3, "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 than causes the fault. See Section 9.3, "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 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 9.3, “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 9.3, “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 6, "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 6.9, "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 9.3, “Task Switching”). If it occurs before the commit point, no program state change occurs. If it occurs after the commit point, the processor will load all the state information from the new TSS (without performing any additional limit, present, or type checks) before it generates the exception. The general-protection exception handler should thus not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. (See the Program State Change description for “Interrupt 10—Invalid TSS Exception (#TS)” in this chapter for additional information on how to handle this situation.)

General Protection Exception in 64-bit Mode

The following conditions cause general-protection exceptions in 64-bit mode:

- If the memory address is in a non-canonical form.
- If a segment descriptor memory address is in non-canonical form.
- If the target offset in a destination operand of a call or jmp is in a non-canonical form.
- If a code segment or 64-bit call gate overlaps non-canonical space.
- If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
- If the EFLAGS.NT bit is set in IRET.
- If the stack segment selector of IRET is null when going back to compatibility mode.
- If the stack segment selector of IRET is null going back to CPL3 and 64-bit mode.
- If a null stack segment selector RPL of IRET is not equal to CPL going back to non-CPL3 and 64-bit mode.
- If the proposed new code segment descriptor of IRET has both the D-bit and the L-bit set.

- If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and it has both the D-bit and the L-bit set.
- If the segment descriptor from a 64-bit call gate is in non-canonical space.
- If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
- If the type field of the upper 64 bits of a 64-bit call gate is not 0.
- If an attempt is made to load a null selector in the SS register in compatibility mode.
- If an attempt is made to load null selector in the SS register in CPL3 and 64-bit mode.
- If an attempt is made to load a null selector in the SS register in non-CPL3 and 64-bit mode where RPL is not equal to CPL.
- If an attempt is made to clear CR0.PG while IA-32e mode is enabled.
- If an attempt is made to set a reserved bit in CR3, CR4 or CR8.

Interrupt 14—Page-Fault Exception (#PF)

Exception Class Fault.

Description

Indicates that, with paging enabled (the PG flag in the CR0 register is set), the processor detected one of the following conditions while using the page-translation mechanism to translate a linear address to a physical address:

- The P (present) flag in a page-directory or page-table entry needed for the address translation is clear, indicating that a page table or the page containing the operand is not present in physical memory.
- The procedure does not have sufficient privilege to access the indicated page (that is, a procedure running in user mode attempts to access a supervisor-mode page). If the SMAP flag is set in CR4, a page fault may also be triggered by code running in supervisor mode that tries to access data at a user-mode address. If either the PKE flag or the PKS flag is set in CR4, the protection-key rights registers may cause page faults on data accesses to linear addresses with certain protection keys.
- Code running in user mode attempts to write to a read-only page. If the WP flag is set in CR0, the page fault will also be triggered by code running in supervisor mode that tries to write to a read-only page.
- An instruction fetch to a linear address that translates to a physical address in a memory page with the execute-disable bit set (for information about the execute-disable bit, see Chapter 5, "Paging"). If the SMEP flag is set in CR4, a page fault will also be triggered by code running in supervisor mode that tries to fetch an instruction from a user-mode address.
- One or more reserved bits in paging-structure entry are set to 1. See description below of RSVD error code flag.
- A shadow-stack access is made to a page that is not a shadow-stack page. See Section 18.2, "Shadow Stacks," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, and Section 5.6, "Access Rights."
- An enclave access violates one of the specified access-control requirements. See Section 36.3, "Access-control Requirements," and Section 36.20, "Enclave Page Cache Map (EPCM)," in Chapter 36, "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 5.7, "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 7-11). The processor establishes the bits in the error code as follows:
  - — P flag (bit 0).
    This flag is 0 if there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address.
  - — W/R (bit 1).
    If the access causing the page-fault exception was a write, this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.
  - — U/S (bit 2).
    If a user-mode access caused the page-fault exception, this flag is 1; it is 0 if a supervisor-mode access did so. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

— RSVD flag (bit 3).
This flag is 1 if there is no translation for the linear address because a reserved bit was set in one of the paging-structure entries used to translate that address.

— I/D flag (bit 4).
This flag is 1 if the access causing the page-fault exception was an instruction fetch. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

— PK flag (bit 5).
This flag is 1 if the access causing the page-fault exception was a data access to a linear address with a protection key for which the protection-key rights registers disallow access.

— SS (bit 6).
If the access causing the page-fault exception was a shadow-stack access (including shadow-stack accesses in enclave mode), this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

— HLAT (bit 7).
This flag is 1 if there is no translation for the linear address using HLAT paging because, in one of the paging-structure entries used to translate that address, either the P flag was 0 or a reserved bit was set. An error code will set this flag only if it clears bit 0 or sets bit 3. This flag will not be set by a page fault resulting from a violation of access rights, nor for one encountered during ordinary paging, including the case in which there has been a restart of HLAT paging.

— SGX flag (bit 15).
This flag is 1 if the exception is unrelated to paging and resulted from violation of SGX-specific access-control requirements. Because such a violation can occur only if there is no ordinary page fault, this flag is set only if the P flag (bit 0) is 1 and the RSVD flag (bit 3) and the PK flag (bit 5) are both 0.

See Section 5.6, “Access Rights,” and Section 5.7, “Page-Fault Exceptions,” for more information about page-fault exceptions and the error codes that they produce.

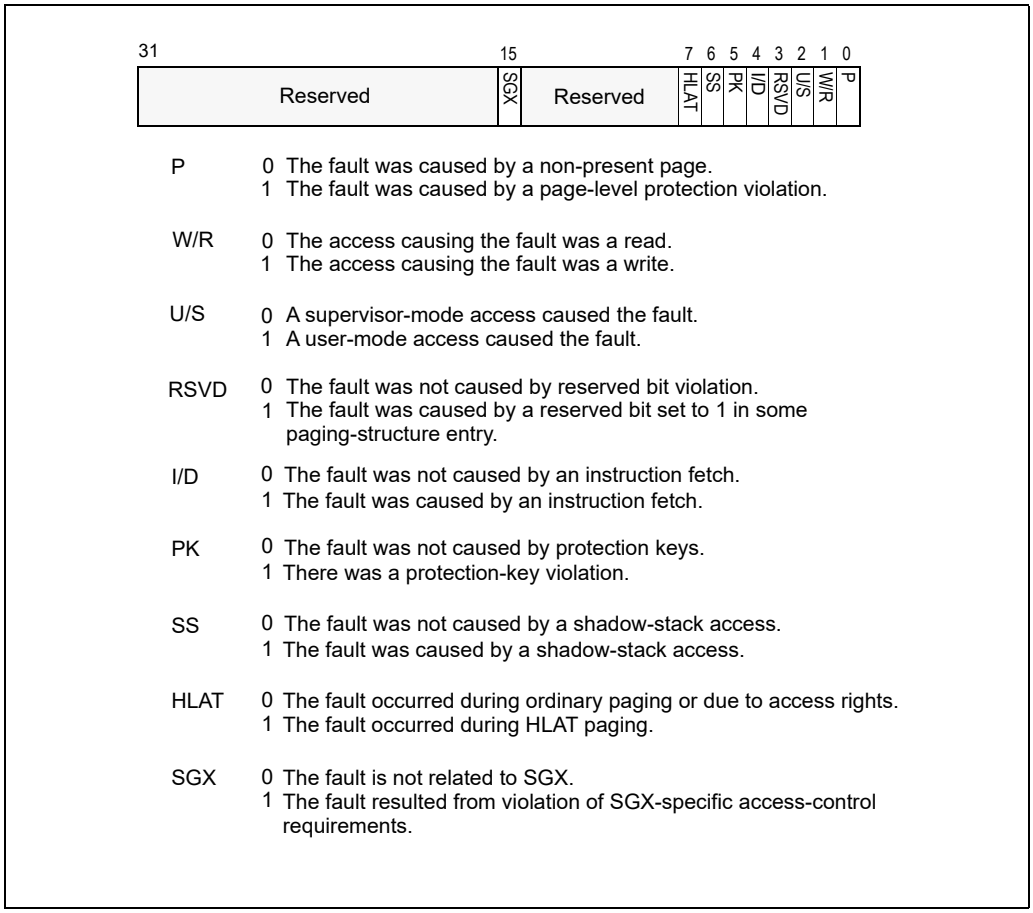


Figure 7-11. Page-Fault Error Code

- The contents of the CR2 register. The processor loads the CR2 register with the linear address that generated the exception. If linear-address masking had been in effect (Section 4.4), the address recorded reflects the result of that masking and does not contain any masked metadata. 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 address are cleared.

  The page-fault handler can use this address to locate the corresponding paging-structure entries. Another page fault can potentially occur during execution of the page-fault handler; the handler should save the contents of the CR2 register before a second page fault can occur.[1] If a page fault is caused by a page-level protection violation, the accessed flags in paging-structure entries may be set when the fault occurs (behavior is model-specific and not architecturally defined).

Saved Instruction Pointer

The saved contents of CS and EIP registers generally point to the instruction that generated the exception. If the page-fault exception occurred during a task switch, the CS and EIP registers may point to the first instruction of the new task (as described in the following "Program State Change" section).

1. Processors update CR2 whenever a page fault is detected. If a second page fault occurs while an earlier page fault is being delivered, the faulting linear address of the second fault will overwrite the contents of CR2 (replacing the previous address). These updates to CR2 occur even if the page fault results in a double fault or occurs during the delivery of a double fault.

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

There are two causes of alignment-check exceptions: alignment violations and bus-lock violations.

Alignment Violations

An **alignment violation** occurs when the processor detects 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 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 7-7 lists the alignment requirements various data types recognized by the processor.

Table 7-7. 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 an alignment violation occurs only for data types that must be aligned on word, doubleword, and quad-word boundaries. A general-protection exception (#GP) is generated 128-bit data types that are not aligned on a 16-byte boundary.

To enable alignment checking, the following conditions must be true:

- AM flag in CR0 register is set.
- AC flag in the EFLAGS register is set.
- The CPL is 3 (including virtual-8086 mode).

Alignment violations 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 violations, 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 violation. 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 alignment violations are enabled when executing these instruc-

tions (and CPL is 3), a misaligned memory operand can cause either an alignment violation 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 6 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D).

The MOVDQU, MOVUPS, and MOVUPD instructions perform 128-bit unaligned loads or stores. The LDDQU instructions 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 violations 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 violations. These instructions are rarely needed by application programs.

Bus -Lock Violations

Some processors include features that disable bus locks. Section 10.1.2.3 provides details. When these features are enabled, occurrence of a bus lock causes a bus-lock violation, leading to a fault. In some cases, the fault is delivered as an alignment-check exception (#AC). The following are the cases in which bus-lock violation leads to an #AC:

- Split-lock disable is enabled (because MSR_MEMORY_CTRL[29] = 1) and locked access to multiple cache lines occurs (a split lock).
- Bus-lock disabled is enabled (because MSR_MEMORY_CTRL[28] = 1), CPUID.(EAX=07H, ECX=2):EDX[bit 6] is enumerated as 1 (indicating support for the architectural form of bus-lock disable), and a locked access using a memory type other than WB occurs (a UC lock).[1]

Exception Error Code

Yes. For alignment violations and bus-lock violations due to split locks, the error code is null, meaning that bits 31:2 of the error code are clear. For bus-lock violations due to UC locks, the error code has value 4, meaning that bit 2 is set.

In either case, bit 0 (EXT) is set if the violation is recognized during delivery of an event other than a software interrupt. (See Section 7.13, "Error Code" and "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.) In such cases, the actual error code delivered will have value 1 or value 5.

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.

1. If CPUID.(EAX=07H, ECX=2):EDX[bit 6] is enumerated as 0, the processor may support an older model-specific form of bus-lock disable. That form generates a general-protection exception (#GP) and not an alignment-check exception.

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 17, "Machine-Check Architecture." Also, see the data books for the individual processors for processor-specific hardware information.

Exception Error Code

None. Error information is provided by machine-check MSRs.

Saved Instruction Pointer

For the Pentium 4 and Intel Xeon processors, the saved contents of extended machine-check state registers are directly associated with the error that caused the machine-check exception to be generated (see Section 17.3.1.2, "IA32_MCG_STATUS MSR," and Section 17.3.2.6, "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 17.3.1.2, "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 OSXMMEXCPT 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 7-8. This exception precedence sometimes results in the higher priority exception condition being reported and the lower priority exception conditions being ignored.

Table 7-8. SIMD Floating-Point Exceptions Priority

Priority	Description
1 (Highest)	Invalid operation exception due to SNaN operand (or any NaN operand for maximum, minimum, or certain compare and convert operations).
2	QNaN operand[1].
3	Any other invalid operation exception not mentioned above or a divide-by-zero exception[2].
4	Denormal operand exception[2].
5	Numeric overflow and underflow exceptions possibly in conjunction with the inexact result exception[2].
6 (Lowest)	Inexact result exception.

NOTES:

1. Though a QNaN this is not an exception, the handling of a QNaN operand has precedence over lower priority exceptions. For example, a QNaN divided by zero results in a QNaN, not a divide-by-zero- exception.
2. If masked, then instruction execution continues, and a lower priority exception can occur as well.

Exception Error Code

None.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the SSE/SSE2/SSE3 instruction that was executed when the SIMD floating-point exception was generated. This is the faulting instruction in which the error condition was detected.

Program State Change

A program-state change does not accompany a SIMD floating-point exception because the handling of the exception is immediate unless the particular exception is masked. The available state information is often sufficient to allow recovery from the error and re-execution of the faulting instruction if needed.

Interrupt 20—Virtualization Exception (#VE)

Exception Class Fault.

Description

Indicates that the processor detected an EPT violation in VMX non-root operation. Not all EPT violations cause virtualization exceptions. See Section 27.5.7.2 for details.

The exception handler can recover from EPT violations and restart the program or task without any loss of program continuity. In some cases, however, the problem that caused the EPT violation may be uncorrectable.

Exception Error Code

None.

Saved Instruction Pointer

The saved contents of CS and EIP registers generally point to the instruction that generated the exception.

Program State Change

A program-state change does not normally accompany a virtualization exception, because the instruction that causes the exception to be generated is not executed. After the virtualization exception handler has corrected the violation (for example, by executing the EPTP-switching VM function), execution of the program or task can be resumed.

Additional Exception-Handling Information

The processor saves information about virtualization exceptions in the virtualization-exception information area. See Section 27.5.7.2 for details.

Interrupt 21—Control Protection Exception (#CP)

Exception Class Fault.

Description

Indicates a control flow transfer attempt violated the control flow enforcement technology constraints.

Exception Error Code

Yes (special format). The processor provides the control protection exception handler with following information through the error code on the stack.

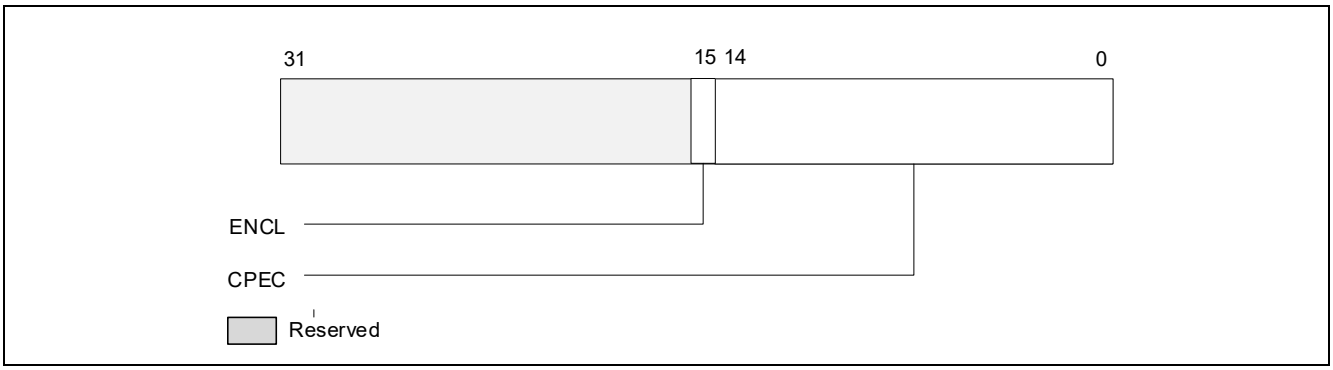


Figure 7-12. Exception Error Code Information

- Bit 14:0 - CPEC
  - 1 - NEAR-RET: Indicates the #CP was caused by a near RET instruction.
  - 2 - FAR-RET/IRET: Indicates the #CP was caused by a FAR RET or IRET instruction.
  - 3 - ENDBRANCH: indicates the #CP was due to missing ENDBRANCH at target of an indirect call or jump instruction.
  - 4 - RSTORSSP: Indicates the #CP was caused by a shadow-stack-restore token check failure in the RSTORSSP instruction.
  - 5- SETSSBSY: Indicates #CP was caused by a supervisor shadow stack token check failure in the SETSSBSY instruction.
- Bit 15 (ENCL) of the error code, if set to 1, indicates the #CP occurred during enclave execution.

Saved Instruction Pointer

The saved contents of the CS and EIP registers generally point to the instruction that generated the exception.

Program State Change

A program-state change does not normally accompany a control protection exception, because the instruction that causes the exception to be generated is not executed.

When a control protection exception is generated during a task switch, the program-state may change as follows. During a task switch, a control protection exception can occur during any of following operations:

- If task switch is initiated by IRET, CS and LIP stored on old task shadow stack do not match CS and LIP of new task (where LIP is the linear address of the return address).
- If task switch is initiated by IRET and SSP of new task loaded from shadow stack of old task (if new task CPL is < 3), OR the SSP from IA32_PL3_SSP (if new task CPL = 3) is not aligned to 4 bytes or is a value beyond 4GB.

In these cases the exception occurs in the context of the new task. The instruction pointer refers to the first instruction of the new task, not to the instruction which caused the task switch (or the last instruction to be executed, in the case of an interrupt). If the design of the operating system permits control protection faults to occur during task-switches, the control protection fault handler should be called through a task gate.

Interrupts 32 to 255—User Defined Interrupts

Exception Class Not applicable.

Description

Indicates that the processor did one of the following things:

- Executed an INT *n* instruction where the instruction operand is one of the vector numbers from 32 through 255.
- Responded to an interrupt request at the INTR pin or from the local APIC when the interrupt vector number associated with the request is from 32 through 255.

Exception Error Code

Not applicable.

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that follows the INT *n* instruction or instruction following the instruction on which the INTR signal occurred.

Program State Change

A program-state change does not accompany interrupts generated by the INT *n* instruction or the INTR signal. The INT *n* instruction generates the interrupt within the instruction stream. When the processor receives an INTR signal, it commits all state changes for all previous instructions before it responds to the interrupt; so, program execution can resume upon returning from the interrupt handler.

CHAPTER 8
USER INTERRUPTS

8.1 INTRODUCTION

This chapter provides details of an architectural feature called **user interrupts**.

This feature defines user interrupts as new events in the architecture. User interrupts are delivered to software operating in 64-bit mode with CPL = 3 without any change to segmentation state. An individual user interrupt is identified by a 6-bit user-interrupt vector, which is pushed on the stack as part of user-interrupt delivery. The UIRET (user-interrupt return) instruction reverses user-interrupt delivery.

System software configures the user-interrupt architecture with MSRs. An operating system (OS) may update the content of some of these MSRs when switching between OS-managed threads.

One of these MSRs references a data structure called the **user posted-interrupt descriptor (UPID)**. User interrupts for an OS-managed thread can be posted in the UPID associated with that thread. Such user interrupts will be delivered after receipt of an ordinary interrupt (identified in the UPID) called a **user-interrupt notification**.[1]

System software can define operations to post user interrupts and to send user-interrupt notifications. In addition, the user-interrupt feature defines the SENDUIPI instruction, by which application software can send interprocessor user interrupts (user IPIs). An execution of SENDUIPI posts a user interrupt in a UPID and may send a user-interrupt notification.

(Platforms may include mechanisms to process external interrupts as either ordinary interrupts or user interrupts. Those processed as user interrupts would be posted in UPIDs and may result in user-interrupt notifications. Specifics of such mechanisms are outside of the scope of this manual.)

Section 8.2 explains how a processor enumerates support for user interrupts and how they are enabled by system software. Section 8.3 identifies the new processor state defined for user interrupts. Section 8.4 explains how a processor identifies and delivers user interrupts. Section 8.5 describes how a processor identifies and processes user-interrupt notifications. Section 8.6 enumerates new instructions that support management of user interrupts. Section 8.8 defines new support for user inter-processor interrupts (user IPIs).

8.2 ENUMERATION AND ENABLING

Software enables user interrupts by setting bit 25 (UINTR) in control register CR4. Setting CR4.UINTR enables user-interrupt delivery (Section 8.4.2), user-interrupt notification identification (Section 8.5.1), and the user-interrupt instructions (Section 8.6). It does not affect the accessibility of the user-interrupt MSRs (Section 8.3) by RDMSR, WRMSR or the XSAVE feature set.

Processor support for user interrupts is enumerated by CPUID.(EAX=7,ECX=0):EDX[5]. If this bit is set, software can set CR4.UINTR to 1 and can access the user-interrupt MSRs using RDMSR and WRMSR (see Section 8.3).

The user-interrupt feature is XSAVE-managed (see Section 13.5). This implies that aspects of the feature are enumerated as part of enumeration of the XSAVE feature set. See Section 13.5.11 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for details.

8.3 USER-INTERRUPT STATE AND USER-INTERRUPT MSRS

The user-interrupt architecture defines the following new state. Some of this state can be accessed via the RDMSR and WRMSR instructions (through new user-interrupt MSRs detailed in Section 8.3.2) and some can be accessed using instructions described in Section 8.6.

1. For clarity, this chapter uses the term **ordinary interrupts** to refer to those events in the existing interrupt architecture, which are typically delivered to system software operating with CPL = 0.

8.3.1 User-Interrupt State

The following are the elements of the user-interrupt state (listed here independent of how they are accessed):

- **UIRR: user-interrupt request register.**

  This value includes one bit for each of the 64 user-interrupt vectors. If UIRR[i] = 1, a user interrupt with vector i is requesting service. The notation **UIRRV** is used to refer to the position of the most significant bit set in UIRR; if UIRR = 0, UIRRV = 0.

- **UIF: user-interrupt flag.**

  If UIF = 0, user-interrupt delivery is blocked; if UIF = 1, user interrupts may be delivered. User-interrupt delivery clears UIF, and the new UIRET instruction sets it. Section 8.6 defines other instructions for accessing UIF and Section 8.7 describes an enhancement that allows UIRET to maintain UIF as 0.

- **UIHANDLER: user-interrupt handler.**

  This is the linear address of the user-interrupt handler. User-interrupt delivery loads this address into RIP.

- **UISTACKADJUST: user-interrupt stack adjustment.**

  This value controls adjustment to the stack pointer (RSP) prior to user-interrupt delivery. It can be configured to load RSP with an alternate stack pointer or configured to prevent user-interrupt delivery from overwriting data above the current stack top.

  The value UISTACKADJUST must be canonical. If bit 0 is 1, user-interrupt delivery loads RSP with UISTACKADJUST; otherwise, it subtracts UISTACKADJUST from RSP. Either way, user-interrupt delivery then aligns RSP to a 16-byte boundary. See Section 8.4.2 for details.

- **UINV: user-interrupt notification vector.**

  This is the vector of the ordinary interrupts that are treated as user-interrupt notifications (Section 8.5.1). When the logical processor receives user-interrupt notification, it processes the user interrupts in the user posted-interrupt descriptor (UPID) referenced by UPIDADDR (see below and Section 8.5.2).

- **UPIDADDR: user posted-interrupt descriptor address.**

  This is the linear address of the UPID that the logical processor consults upon receiving an ordinary interrupt with vector UINV.

- **UITTADDR: user-interrupt target table address.**

  This is the linear address of user-interrupt target table (UITT), which the logical processor consults when software executes the SENDUIPI instruction (see Section 8.8).

- **UITTSZ: user-interrupt target table size.**

  This value is the highest index of a valid entry in the UITT (see Section 8.8).

8.3.2 User-Interrupt MSRs

Some of the state elements identified in Section 8.3.1 can be accessed as user-interrupt MSRs using the RDMSR and WRMSR instructions:

- IA32_UINTR_RR MSR (MSR address 985H). This MSR is an interface to UIRR (64 bits).

  Following a WRMSR to this MSR, the logical processor recognizes a pending user interrupt if and only if some bit is set in the MSR.

- IA32_UINTR_HANDLER MSR (MSR address 986H). This MSR is an interface to the UIHANDLER address. This is a linear address that must be canonical relative to the maximum linear-address width supported by the processor.[1] WRMSR to this MSR causes a general-protection fault (#GP) if its source operand does not meet this requirement.

- IA32_UINTR_STACKADJUST MSR (MSR address 987H). This MSR is an interface to the UISTACKADJUST value. This value includes a linear address that must be canonical relative to the maximum linear-address width supported by the processor. WRMSR to this MSR causes a #GP if its source operand does not meet this requirement.

1. CPUID.80000008H:EAX[15:8] enumerates the maximum linear-address width supported by the processor.

Bit 0 of this MSR corresponds to UISTACKADJUST[0], which controls how user-interrupt delivery updates the stack pointer. WRMSR may set it to either 0 or 1.

- IA32_UINTR_MISC MSR (MSR address 988H). This MSR is an interface to the UITTSZ and UINV values. The MSR has the following format:
  - — Bits 31:0 are UITTSZ.
  - — Bits 39:32 are UINV.
  - — Bits 63:40 are reserved. WRMSR causes a #GP if it would set any of those bits (if EDX[31:8] ≠ 000000H).

  Because this MSR will share an 8-byte portion of the XSAVE area with UIF (see Section 13.5.11 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1), bit 63 of the MSR will never be used and will always be reserved.
- IA32_UINTR_PD MSR (MSR address 989H). This MSR is an interface to the UPIDADDR address. This is a linear address that must be canonical relative to the maximum linear-address width supported by the processor. WRMSR to this MSR causes a #GP if its source operand does not meet this requirement.

  Bits 5:0 of this MSR are reserved. WRMSR causes a #GP if it would set any of those bits (if EAX[5:0] ≠ 000000b).
- IA32_UINTR_TT MSR (MSR address 98AH). This MSR is an interface to the UITTADDR address (in addition, bit 0 enables SENDUIPI).

  Bit 63:4 of this MSR holds the current value of UITTADDR. This a linear address that must be canonical relative to the maximum linear-address width supported by the processor. WRMSR to this MSR causes a #GP if its source operand does not meet this requirement.

  Bits 3:1 of this MSR are reserved. WRMSR causes a #GP if it would set any of those bits (if EAX[3:1] ≠ 000b).

  Bit 0 of this MSR determines whether the SENDUIPI instruction is enabled. WRMSR may set it to either 0 or 1.

8.4 EVALUATION AND DELIVERY OF USER INTERRUPTS

A processor determines whether there is a user interrupt to deliver based on UIRR. Section 8.4.1 describes this recognition of pending user interrupts. Once a logical processor has recognized a pending user interrupt, it will deliver it on a subsequent instruction boundary by causing a control-flow change asynchronous to software execution. Section 8.4.2 details this process of user-interrupt delivery.

8.4.1 User-Interrupt Recognition

There is a user interrupt pending whenever UIRR ≠ 0.

Any instruction or operation that modifies UIRR updates the logical processor's recognition of a pending user interrupt. The following instructions and operations may do this:

- WRMSR to the IA32_UINTR_RR MSR (Section 8.3).
- XRSTORS of the user-interrupt state component.
- User-interrupt delivery (Section 8.4.2).
- User-interrupt notification processing (Section 8.5.2).
- VMX transitions that load the IA32_UINTR_RR MSR.

Each of these instructions or operations results in recognition of a pending user interrupt if it completes with UIRR ≠ 0; if it completes with UIRR = 0, no pending user interrupt is recognized.

Once recognized, a pending user interrupt may be delivered to software; see Section 8.4.2.

8.4.2 User-Interrupt Delivery

If CR4.UINTR = 1 and a user interrupt has been recognized (see Section 8.4.1), it will be delivered at an instruction boundary when the following conditions all hold: (1) UIF = 1; (2) there is no blocking by MOV SS or by POP SS[1]; (3) CPL = 3; (4) IA32_EFER.LMA = CS.L = 1 (the logical processor is in 64-bit mode); and (5) software is not executing inside an enclave.

User-interrupt delivery has priority just below that of ordinary interrupts. It wakes a logical processor from the states entered using the TPAUSE and UMWAIT instructions[2]; it does not wake a logical processor in the shutdown state or in the wait-for-SIPI state.

User-interrupt delivery does not change CPL (it occurs entirely with CPL = 3). The following pseudocode details the behavior of user-interrupt delivery:

```
IF UIHANDLER is not canonical in current paging mode
      THEN #GP(0);
FI;
holdRSP := RSP;
IF UISTACKADJUST[0] = 1
      THEN RSP := UISTACKADJUST;
      ELSE RSP := RSP – UISTACKADJUST;
FI;
RSP := RSP & ~FH;                  // force the stack to be 16-byte aligned
Push holdRSP;
Push RFLAGS;
Push RIP;
Push UIRRV;                        // 64-bit push; upper 58 bits pushed as 0
IF shadow stack is enabled for CPL = 3
      THEN ShadowStackPush RIP;
FI;
IF end-branch is enabled for CPL = 3
      THEN IA32_U_CET.TRACKER := WAIT_FOR_ENDBRANCH;
FI;
UIRR[Vector] := 0;
IF UIRR = 0
      THEN cease recognition of any pending user interrupt;
FI;
UIF := 0;
RFLAGS.TF := 0;
RFLAGS.RF := 0;
RIP := UIHANDLER;
```

If UISTACKADJUST[0] = 0, user-interrupt delivery decrements RSP by UISTACKADJUST; otherwise, it loads RSP with UISTACKADJUST. In either case, user-interrupt delivery aligns RSP to a 16-byte boundary by clearing RSP[3:0].

User-interrupt delivery that occurs during transactional execution causes transactional execution to abort and a transition to a non-transactional execution. The transactional abort loads EAX as it would had it been due to an ordinary interrupt. User-interrupt delivery occurs after the transactional abort.

---

1. Execution of the STI instruction does not block delivery of user interrupts for one instruction as it does ordinary interrupts. If a user interrupt is delivered immediately following execution of a STI instruction, ordinary interrupts are not blocked after delivery of the user interrupt.
2. User-interrupt delivery occurs only if CPL = 3. Since the HLT and MWAIT instructions can be executed only if CPL = 0, a user interrupt can never be delivered when a logical processor is an activity state that was entered using one of those instructions.

The stack accesses performed by user-interrupt delivery may incur faults (page faults, or stack faults due to canonicality violations). Before such a fault is delivered, RSP is restored to its original value (memory locations above the top of the stack may have been written). If such a fault produces an error code that uses the EXT bit, that bit will be cleared to 0.

If a fault occurs during user-interrupt delivery, UIRR is not updated and UIF is not cleared and, as a result, the logical processor continues to recognize that a user interrupt is pending, and user-interrupt delivery will normally recur after the fault is handled.

If the shadow-stack feature of control-flow enforcement technology (CET) is enabled for CPL = 3, user-interrupt delivery pushes the return instruction pointer on the shadow stack. If indirect-branch-tracking feature of CET is enabled, user-interrupt delivery transitions the indirect branch tracker to the WAIT_FOR_ENDBRANCH state; an ENDBR64 instruction is expected as first instruction of the user-interrupt handler.

User-interrupt delivery can be tracked by Architectural Last Branch Records (LBRs), Intel® Processor Trace (Intel® PT), and Performance Monitoring. For both Intel PT and LBRs, user-interrupt delivery is recorded in precisely the same manner as ordinary interrupt delivery. Hence for LBRs, user interrupts fall into the OTHER_BRANCH category, which implies that IA32_LBR_CTL.OTHER_BRANCH[bit 22] must be set to record user-interrupt delivery, and that the IA32_LBR_x_INFO.BR_TYPE field will indicate OTHER_BRANCH for any recorded user interrupt. For Intel PT, control flow tracing must be enabled by setting IA32_RTIT_CTL.BranchEn[bit 13].

User-interrupt delivery will also increment performance counters for which counting BR_INST_RETIRED.FAR_BRANCH is enabled. Some implementations may have dedicated events for counting user-interrupt delivery; see processor-specific event lists at https://download.01.org/perfmon/index/.

8.5 USER-INTERRUPT NOTIFICATION IDENTIFICATION AND PROCESSING

User-interrupt posting is the process by which a platform agent (or software operating on a CPU) records user interrupts in a **user posted-interrupt descriptor** (UPID) in memory. The platform agent (or software) may send an ordinary interrupt (called a **user-interrupt notification**) to the logical processor on which the target of the user interrupt is operating.

Table 8-1 gives the format of a UPID.

Table 8-1. Format of User Posted-Interrupt Descriptor — UPID

Bit Position(s)	Name	Description
0	Outstanding notification	If this bit is set, there is a notification outstanding for one or more user interrupts in PIR.
1	Suppress notification	If this bit is set, agents (including SENDUIPI) should not send notifications when posting user interrupts in this descriptor.
15:2	Reserved	User-interrupt notification processing ignores these bits; must be zero for SENDUIPI.
23:16	Notification vector	Used by agents sending user-interrupt notifications (including SENDUIPI).
31:24	Reserved	User-interrupt notification processing ignores these bits; must be zero for SENDUIPI.
63:32	Notification destination	Target physical APIC ID – used by SENDUIPI. In xAPIC mode, bits 47:40 are the 8-bit APIC ID. In x2APIC mode, the entire field forms the 32-bit APIC ID.
127:64	Posted-interrupt requests (PIR)	One bit for each user-interrupt vector. There is a user-interrupt request for a vector if the corresponding bit is 1.

The notation **PIR** (posted-interrupt requests) refers to the 64 posted-interrupt requests in a UPID.

If an ordinary interrupt arrives while CR4.UINTR = IA32_EFER.LMA = 1, the logical processor determines whether the interrupt is a user-interrupt notification. This process is called **user-interrupt notification identification** and is described in Section 8.5.1.

Once a logical processor has identified a user-interrupt notification, it copies user interrupts in the UPID's PIR into UIRR. This process is called **user-interrupt notification processing** and is described in Section 8.5.2.

A logical processor is not interruptible during either user-interrupt notification identification or user-interrupt notification processing or between those operations (when they occur in succession).

8.5.1 User-Interrupt Notification Identification

If CR4.UINTR = IA32_EFER.LMA = 1, a logical processor performs user-interrupt notification identification when it receives an ordinary interrupt. The following algorithm describes the response by the processor to an ordinary maskable interrupt when CR4.UINTR = IA32_EFER.LMA = 1[1]:

1. The local APIC is acknowledged; this provides the processor core with an interrupt vector, V.
2. If V = UINV, the logical processor continues to the next step. Otherwise, an interrupt with vector V is delivered normally through the IDT; the remainder of this algorithm does not apply and user-interrupt notification processing does not occur.
3. The processor writes zero to the EOI register in the local APIC; this dismisses the interrupt with vector V = UINV from the local APIC.

User-interrupt notification identification involves acknowledgment of the local APIC and thus occurs only when ordinary interrupts are not masked.

If user-interrupt notification identification completes step #3, the logical processor then performs user-interrupt notification processing as described in Section 8.5.2.

An ordinary interrupt that occurs during transactional execution causes the transactional execution to abort and transition to a non-transactional execution. This occurs before user-interrupt notification identification.

An ordinary interrupt that occurs while software is executing inside an enclave causes an asynchronous enclave exit (AEX). This AEX occurs before user-interrupt notification identification.

8.5.2 User-Interrupt Notification Processing

Once a logical processor has identified a user-interrupt notification, it performs **user-interrupt notification processing** using the UPID at the linear address in the IA32_UINTR_PD MSR.

The following algorithm describes user-interrupt notification processing:

1. The logical processor clears the outstanding-notification bit (bit 0) in the UPID. This is done atomically so as to leave the remainder of the descriptor unmodified.
2. The logical processor reads PIR (bits 127:64 of the UPID) into a temporary register and writes all zeros to PIR. This is done atomically so as to ensure that each bit cleared in PIR is set in the temporary register.
3. If any bit is set in the temporary register, the logical processor sets in UIRR each bit corresponding to a bit set in the temporary register (e.g., with a logical OR) and recognizes a pending user interrupt (if it has not already done so).

The logical processor performs the steps above in an uninterruptible manner. Steps #1 and #2 may be combined into a single atomic step. If step #3 leads to recognition of a user interrupt, the processor may deliver that user interrupt on the following instruction boundary (see Section 8.4.2).

Although user-interrupt notification processing may occur at any privilege level, all of the memory accesses in steps #1 and #2 are performed with supervisor privilege.

Step #1 and step #2 each access the UPID using a linear address and may therefore incur faults (page faults, or general-protection faults due to canonicality violations). If such a fault produces an error code that uses the EXT bit, that bit will be set to 1.

If a fault occurs during user-interrupt notification processing, updates to architectural state performed by the earlier user-interrupt notification identification (Section 8.5.1) remain committed and are not undone; if such a fault occurs at step #2 (if it is not performed atomically with step #1), any update to architectural state performed by step #1 also remains committed. System software is advised to prevent such faults (e.g., by ensuring that no

1. If the interrupt arrives between iterations of a REP-prefixed string instruction, the processor first updates state as follows: RIP is loaded to reference the string instruction; RCX, RSI, and RDI are updated as appropriate to reflect the iterations completed; and RFLAGS.RF is set to 1.

page fault occurs and that the linear address in the IA32_UINTR_PD MSR is canonical with respect to the paging mode in use).

If the user-interrupt notification identification that precedes user-interrupt notification processing occurred due to an ordinary interrupt that arrived while the logical processor was in the HLT state, the logical processor returns to the HLT state following user-interrupt notification processing.

8.6 USER-INTERRUPT INSTRUCTIONS

The user-interrupt feature defines instructions for control-flow transfer and access to new state. UIRET is an instruction to effect a return from a user-interrupt handler. CLUI, STUI, and TESTUI allow software to access UIF. SENDUIPI sends a user IPI. See Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D for details on the instructions' operation.

The following items provide high-level overviews of the instructions:

- UIRET pops from the stack the state saved by user-interrupt delivery (see Section 8.4.2) and loads those values into the corresponding registers (software should pop the user-interrupt vector from the stack before executing UIRET). Because RIP is one of those registers, UIRET effect a return to the that point from which the user interrupt was delivered.
- CLUI clears UIF.
- STUI sets UIF.
- TESTUI copies UIF to RFLAGS.CF.
- SENDUIPI is discussed in Section 8.8.

8.7 FLEXIBLE UPDATES OF UIF BY UIRET

There may be software usages that seek to return from the handling of a user interrupt while maintaining UIF as 0. This section describes an enhancement to UIRET that allows that.

This enhancement is supported if CPUID.(EAX=07H, ECX=01H):EDX.UIRET_UIF[bit 17] is enumerated as 1.

If the enhancement is supported, UIRET loads UIF with the value of the bit at position 1 in the RFLAGS image on the stack. (If the enhancement is not supported, UIRET ignores that bit in the RFLAGS image and always sets UIF to 1.)

The value of RFLAGS[1] is fixed as 1. All operations that save RFLAGS to memory save bit 1 as set; all operations that load RFLAGS leave bit 1 set.

This implies that user-interrupt delivery always saves on the stack an RFLAGS value that sets bit 1. If software does not modify this stack value, UIRET will set UIF to 1, even with the enhancement. Thus, the enhancement should not affect the operation of existing software, as long as that software does not modify the stack value saved for RFLAGS[1].

If a user-interrupt handler seek to return from the handling of a user interrupt while maintaining UIF as 0, it should modify the RFLAGS image on the stack to clear bit 1. Subsequent execution of UIRET will load UIF with 0, as long as the enhancement is supported.

Note that UIRET never modifies RFLAGS[1] (always leaving it with value 1) regardless of the stack value and regardless of whether enhancement is supported.

See Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D for details on the operation of the UIRET instruction.

8.8 USER IPIS

The SENDUIPI instruction sends a user interprocessor interrupt (IPI). The instruction uses a data structure called the user-interrupt target table (UITT). This table is located at the linear address UITTADDR and it comprises

UITTSZ+1 16-byte entries (the values UITTADDR and UITTSZ are defined in Section 8.3.1). SENDUIPI uses the UITT entry (UITTE) indexed by the instruction's register operand. Each UITTE has the following format:

- Bit 0: V, a valid bit.
- Bits 7:1 are reserved and must be 0.
- Bits 15:8: UV, the user-interrupt vector (in the range 0–63, so bits 15:14 must be 0).
- Bits 63:16 are reserved.
- Bits 127:64: UPIDADDR, the linear address of a UPID (64-byte aligned, so bits 69:64 must be 0).

SENDUIPI sends a user interrupt by posting a user interrupt with vector V in the UPID referenced by UPIDADDR and then sending, as an ordinary IPI, any notification interrupt specified in that UPID. Details appear in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

CHAPTER 9
TASK MANAGEMENT

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 9.6, "16-Bit Task-State Segment (TSS)." For information specific to task management in 64-bit mode, see Section 9.7, "Task Management in 64-bit Mode."

9.1 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.

9.1.1 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 9-1). 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.

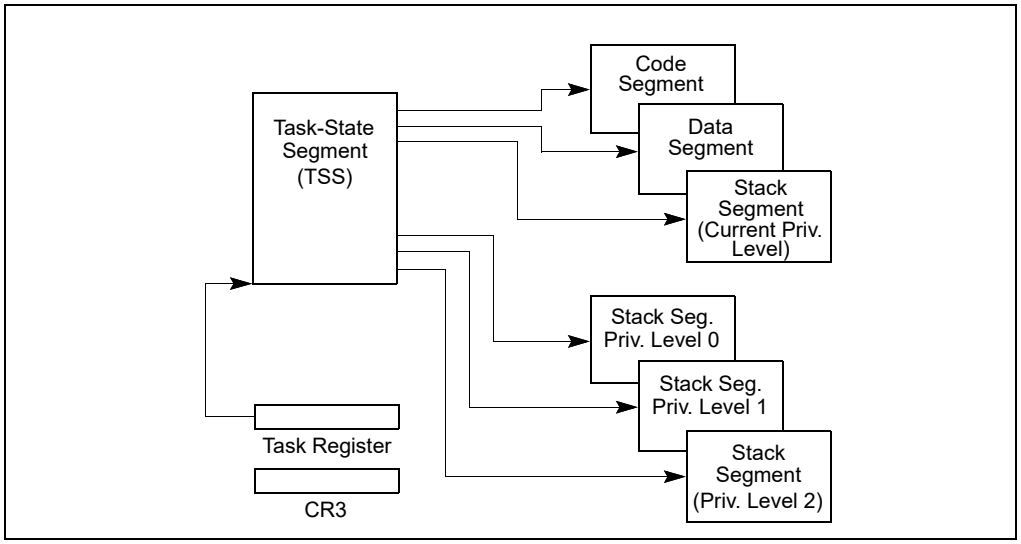


Figure 9-1. Structure of a Task

9.1.2 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).
- The state of the shadow stack pointer (SSP).

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.

9.1.3 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.

If shadow stack is enabled, then the SSP of the task is located at the 4 bytes at offset 104 in the 32-bit TSS and is used by the processor to establish the SSP when a task switch occurs from a task associated with this TSS. Note that the processor does not write the SSP of the task initiating the task switch to the TSS of that task, and instead the SSP of the previous task is pushed onto the shadow stack of the new task.

9.2 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).

9.2.1 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 9-2 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 9.6, "16-Bit Task-State Segment (TSS)." For information about 64-bit mode task structures, see Section 9.7, "Task Management in 64-bit Mode."

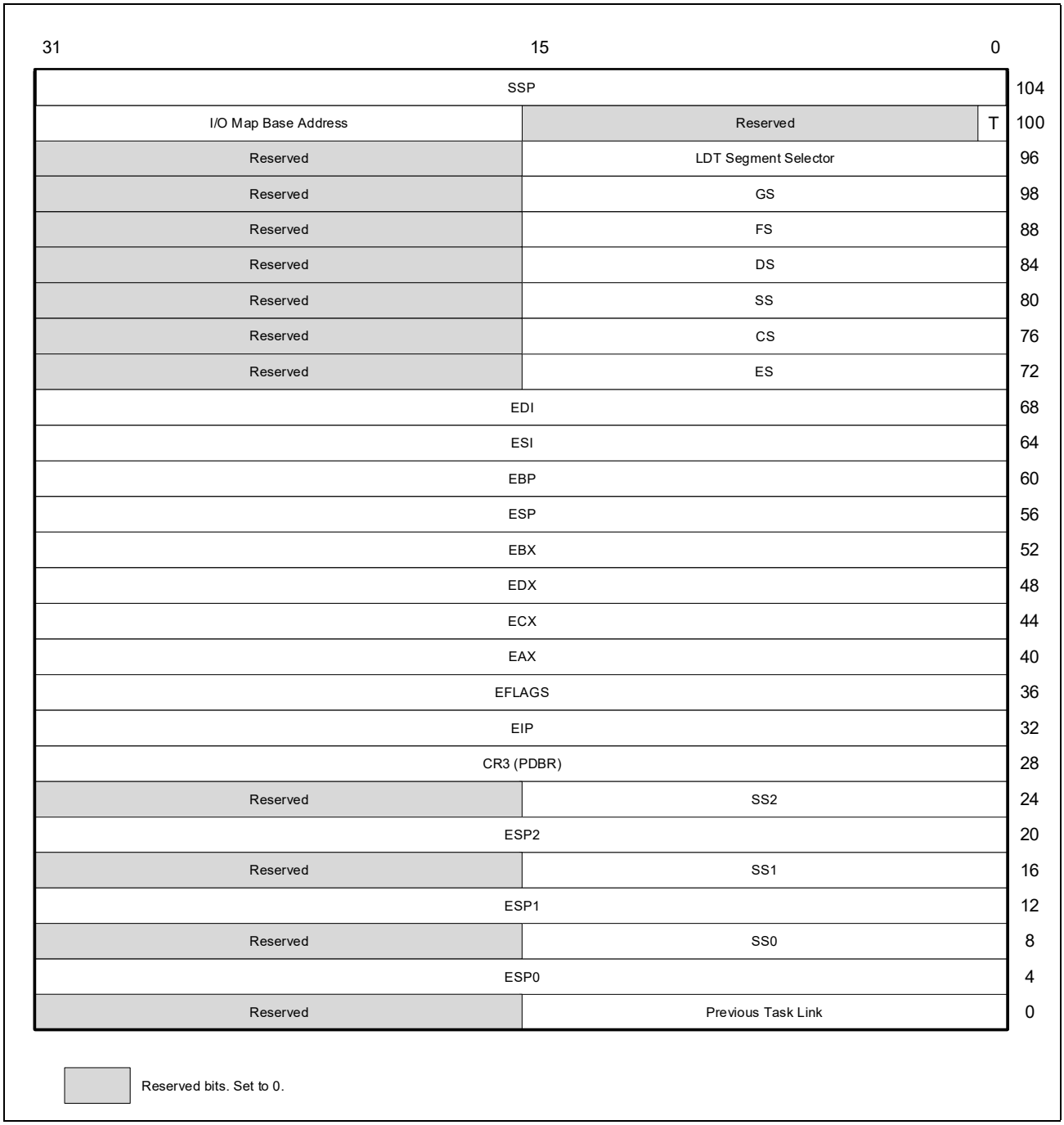


Figure 9-2. 32-Bit Task-State Segment (TSS)

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 EFLAGS 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 19.3.1.5, "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 20, "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 22.3, "Interrupt and Exception Handling in Virtual-8086 Mode," for a detailed description of the interrupt redirection bit map.
- **Shadow Stack Pointer (SSP)** — Contains task's shadow stack pointer. The shadow stack of the task should have a supervisor shadow stack token at the address pointed to by the task SSP (offset 104). This token will be verified and made busy when switching to that shadow stack using a CALL/JMP instruction, and made free when switching out of that task using an IRET instruction.

If paging is used:

- 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.

9.2.2 TSS Descriptor

The TSS, like all other segments, is defined by a segment descriptor. Figure 9-3 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 ensure that there is only one busy flag is associated with a task, each TSS should have only one TSS descriptor that points to it.

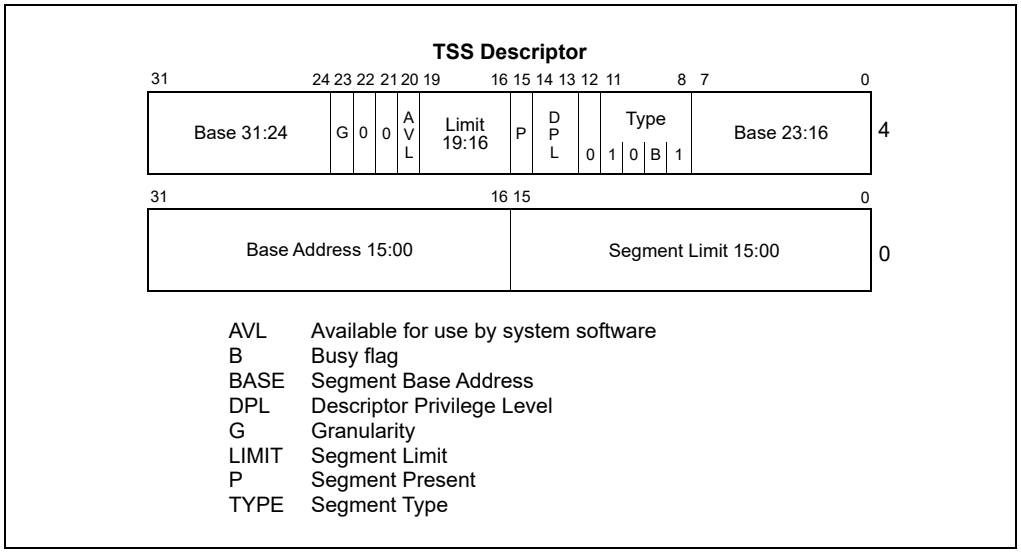


Figure 9-3. TSS Descriptor

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.

9.2.3 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 9.7.

In 64-bit mode, the TSS descriptor is expanded to 16 bytes (see Figure 9-4). This expansion also applies to an LDT descriptor in 64-bit mode. Table 3-2 provides the encoding information for the segment type field.

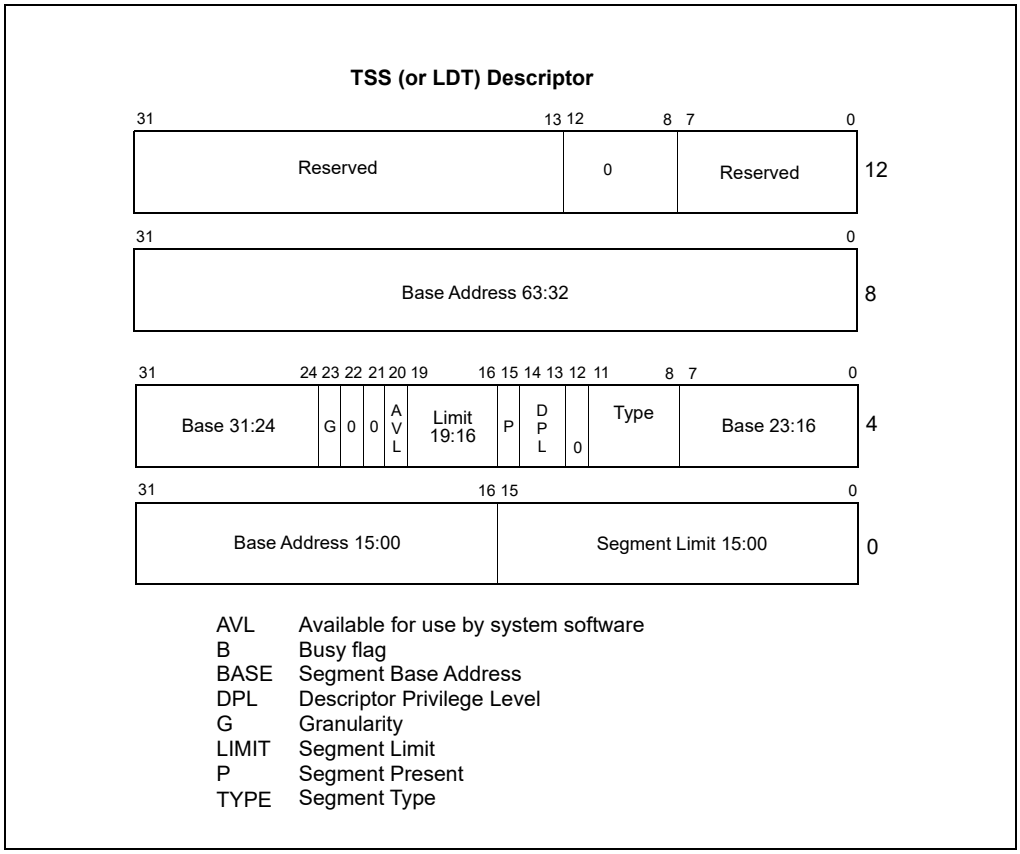


Figure 9-4. Format of TSS and LDT Descriptors in 64-bit Mode

9.2.4 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 9-5 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.

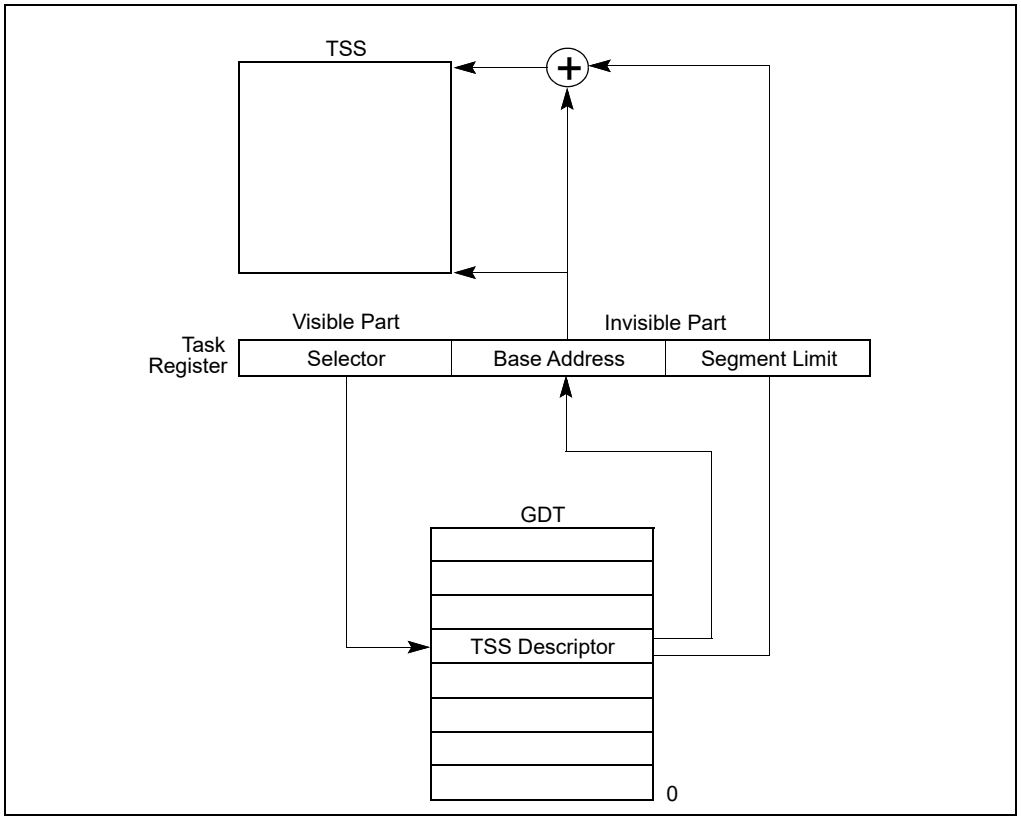


Figure 9-5. Task Register

9.2.5 Task-Gate Descriptor

A task-gate descriptor provides an indirect, protected reference to a task (see Figure 9-6). 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.

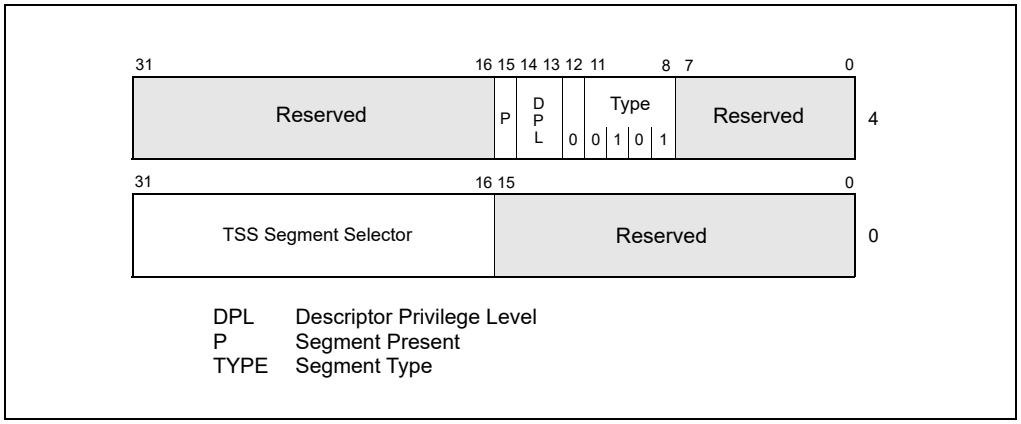


Figure 9-6. Task-Gate Descriptor

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 fla**g — 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 9-7 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.

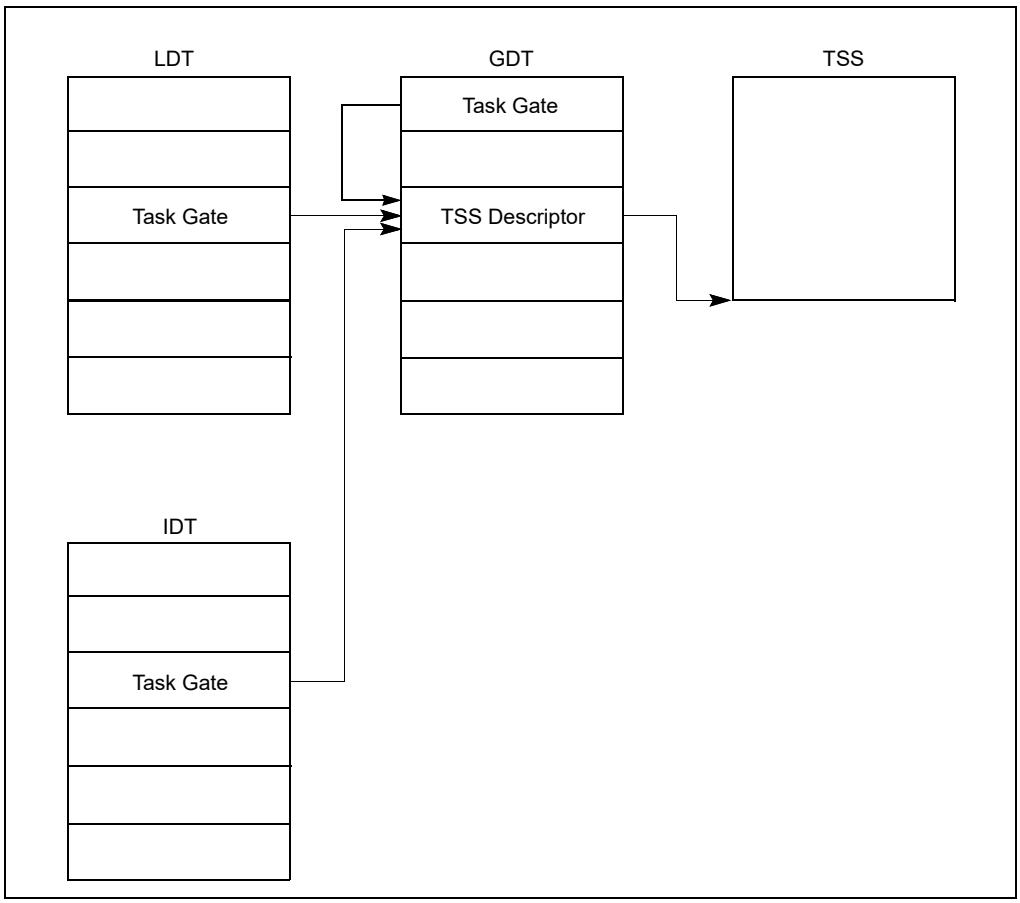


Figure 9-7. Task Gates Referencing the Same Task

9.3 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.[1]
3. Checks that the TSS descriptor of the new task is marked present and has a valid limit (greater than or equal to 67H). If the task switch was initiated by IRET and shadow stacks are enabled at the current CPL, then the SSP must be aligned to 8 bytes, else a #TS(current task TSS) fault is generated. If CR4.CET is 1, then the TSS must be a 32 bit TSS and the limit of the new task's TSS must be greater than or equal to 107 bytes, else a #TS(new task TSS) fault is generated.
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. 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).
7. Loads the task register with the segment selector and descriptor for the new task's TSS.
8. If CET is enabled, the processor performs following shadow stack actions:

```
Read CS of new task from new task TSS
Read EFLAGS of new task from new task TSS
IF EFLAGS.VM = 1
    THEN
        new task CPL = 3;
    ELSE
        new task CPL = CS.RPL;
FI;
pushCsLipSsp = 0
IF task switch was initiated by CALL instruction, exception or interrupt
    IF shadow stack enabled at current CPL
        IF new task CPL < CPL and current task CPL = 3
            THEN
                IA32_PL3_SSP = SSP (* user ⟶ supervisor *)
            ELSE
                pushCsLipSsp = 1 (* no privilege change; supv ⟶ supv; supv ⟶ user *) tempSSP = SSP
```

1. The INT1 has opcode F1; the INT *n* instruction with *n*=1 has opcode CD 01.

```
                tempSsLIP =CSBASE + EIP
                tempSsCS = CS
            FI;
        FI;
    FI;
    verifyCsLIP = 0
    IF task switch was initiated by IRET
        IF shadow stacks enabled at current CPL
            IF (CPL of new Task = CPL of current Task) OR
                (CPL of new Task < 3 AND CPL of current Task < 3) OR
                (CPL or new Task < 3 AND CPL of current task = 3)
                    (* no privilege change or supervisor → supervisor or user → supervisor IRET *)
                    tempSsCS = shadow_stack_load 8 bytes from SSP+16;
                    tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
                    tempSSP = shadow_stack_load 8 bytes from SSP;
                    SSP = SSP + 24;
                    verifyCsLIP = 1
            FI;
            // Clear busy flag on current shadow stack
             IF ( SSP & 0x07 == 0 )                    (* SSP must be aligned to 8B *)
                THEN
                    expected_token_value = (SSP & ~0x07) | BUSY_BIT;  (* busy - bit 0 - must be set*)
                    new_token_value      = SSP                        (* clear the busy bit *)
                    shadow_stack_lock_cmpxchg8b(SSP, new_token_value, expected_token_value)
            FI;
            SSP = 0
        FI;
    FI;
```

9. 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.)
10. 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 9-2.)
11. 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.
12. 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 9-2).
13. 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.
14. 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.

15. If CET is enabled, the processor performs following shadow stack actions:

```
IF shadow stack enabled at current CPL OR indirect branch tracking at current CPL
        THEN
          IF EFLAGS.VM = 1
            THEN #TSS(new-Task-TSS);FI;
FI;
IF shadow stack enabled at current CPL
        IF task switch initiated by CALL instruction, JMP instruction, interrupt or exception (* switch stack *)
          new_SSP ← Load the 4 byte from offset 104 in the TSS
          // Verify new SSP to be legal
          IF new_SSP & 0x07 != 0
            THEN #TSS(New-Task-TSS); FI;
          expected_token_value = SSP;                  (* busy - bit 0 - must be clear *)
          new_token_value        = SSP | BUSY_BIT (* set the busy bit - bit 0*)
          IF shadow_stack_lock_cmpxchg8b(SSP, new_token_value,
                                                expected_token_value) != expected_token_value
            THEN #TSS(New-Task-TSS); FI;
          SSP = new_SSP
          IF pushCsLipSsp = 1 (* call, int, exception from user ⟶ user or supv ⟶ supv or supv ⟶ user *)
            Push tempSsCS, tempSsLip, tempSsSSP on shadow stack using 8B pushes[1]
          FI;
        FI;
FI;
IF task switch initiated by IRET
        IF verifyCsLIP = 1
          (* do 64 bit comparisons; CS zero padded to 64 bit; CSBASE+EIP zero padded to 64 bit *)
          IF tempSsCS and tempSsLIP do not match CS and CSBASE+EIP
            THEN #CP(FAR-RET/IRET); FI;
        FI;
        IF ShadowStackEnabled(CPL)
          THEN
            IF (verifyCsLIP == 0) tempSSP = IA32_PL3_SSP;
              IF tempSSP & 0x03 != 0 THEN #CP(FAR-RET/IRET) // verify aligned to 4 bytes
              IF tempSSP[63:32] != 0 THEN # CP(FAR-RET/IRET)
            SSP = tempSSP
        FI;
FI;
IF EndbranchEnabled(CPL)
        IF task switch initiated by CALL instruction, JMP instruction, interrupt or exception
          IF CPL = 3
            THEN
              IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
```

1. If any of these pushes leads to an exception or a VM exit, the supervisor shadow-stack token remains busy.

```
            IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
            IA32_S_CET.SUPPRESS = 0
    FI;
  FI;
FI;
```

16. Begins executing the new task. (To an exception handler, the first instruction of the new task appears not to have been executed.)

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 8, the processor does not complete the task switch and ensures that the processor is returned to its state prior to the execution of the instruction that initiated the task switch.

If an unrecoverable error occurs in step 9, 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 7, "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.

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 9-1 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 9-1. Exception Conditions Checked During a Task Switch

Condition Checked	Exception[1]	Error Code Reference[2]
Segment selector for a TSS descriptor references the GDT and is within the limits of the table.	#GP #TS (for IRET)	New Task's TSS
P bit is set in TSS descriptor.	#NP	New Task's TSS
TSS descriptor is not busy (for task switch initiated by a call, interrupt, or exception).	#GP (for JMP, CALL, INT)	Task's back-link TSS
TSS descriptor is not busy (for task switch initiated by an IRET instruction).	#TS (for IRET)	New Task's TSS
TSS segment limit greater than or equal to 108 (for 32-bit TSS) or 44 (for 16-bit TSS).	#TS	New Task's TSS

Table 9-1. Exception Conditions Checked During a Task Switch (Contd.)

Condition Checked	Exception[1]	Error Code Reference[2]
TSS segment limit greater than or equal to 108 (for 32-bit TSS) if CR4.CET = 1.[3]	#TS	New Task's TSS
If shadow stack enabled and SSP not aligned to 8 bytes (for task switch initiated by an IRET instruction).[3]	#TS	Current Task's TSS
Registers are loaded from the values in the TSS.		
LDT segment selector of new task is valid [4].	#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 [2].	#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 [4].	#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 [4].	#TS	New Data Segment
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
Shadow Stack Pointer in a task not aligned to 8 bytes (for task switch initiated by a call, interrupt, or exception).[3]	#TS	New Task's TSS
If EFLAGS.VM=1 and shadow stacks are enabled.[3]	#TS	New Task's TSS
Supervisor Shadow Stack Token verification failures (for task switch initiated by a call, interrupt, jump, or exception):[3]	#TS	New Task's TSS
- Busy bit already set.		
- Address in Shadow stack token does not match SSP value from TSS.		
If task switch initiated by IRET, CS and LIP stored on old task shadow stack does not match CS and LIP of new task.[3]	#CP	FAR-RET/IRET
If task switch initiated by IRET and SSP of new task loaded from shadow stack of old task (if new task CPL is < 3) OR the SSP from IA32_PL3_SSP (if new task CPL = 3) fails the following checks:[3]	#CP	FAR-RET/IRET
- Not aligned to 4 bytes.		
- Is beyond 4G.		

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. Valid when CET is enabled.
4. 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.

9.4 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 9-8.

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.

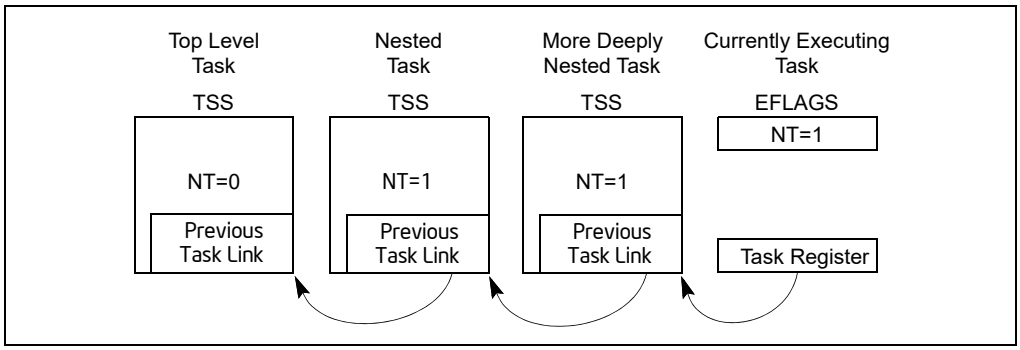


Figure 9-8. Nested Tasks

Table 9-2 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 9-2. 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.

9.4.1 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 10.1.2.1, "Automatic Locking," for more information about setting the busy flag in a multiprocessor applications.

9.4.2 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 remove must be cleared.
4. Enable interrupts.

In a multiprocessing system, additional synchronization and serialization operations must be added to this procedure to ensure that the TSS and its segment descriptor are both locked when the previous task link field is changed and the busy flag is cleared.

9.5 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.

9.5.1 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 9-9 shows how the linear address spaces of two tasks can overlap in the physical space by sharing page tables.

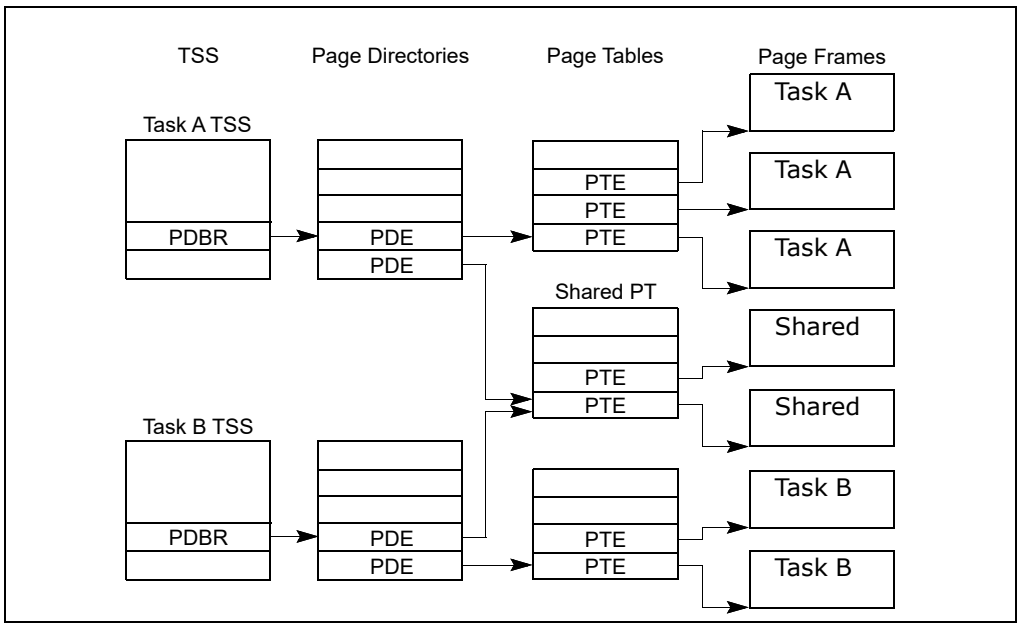


Figure 9-9. Overlapping Linear-to-Physical Mappings

9.5.2 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.

9.6 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 9-10). 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.

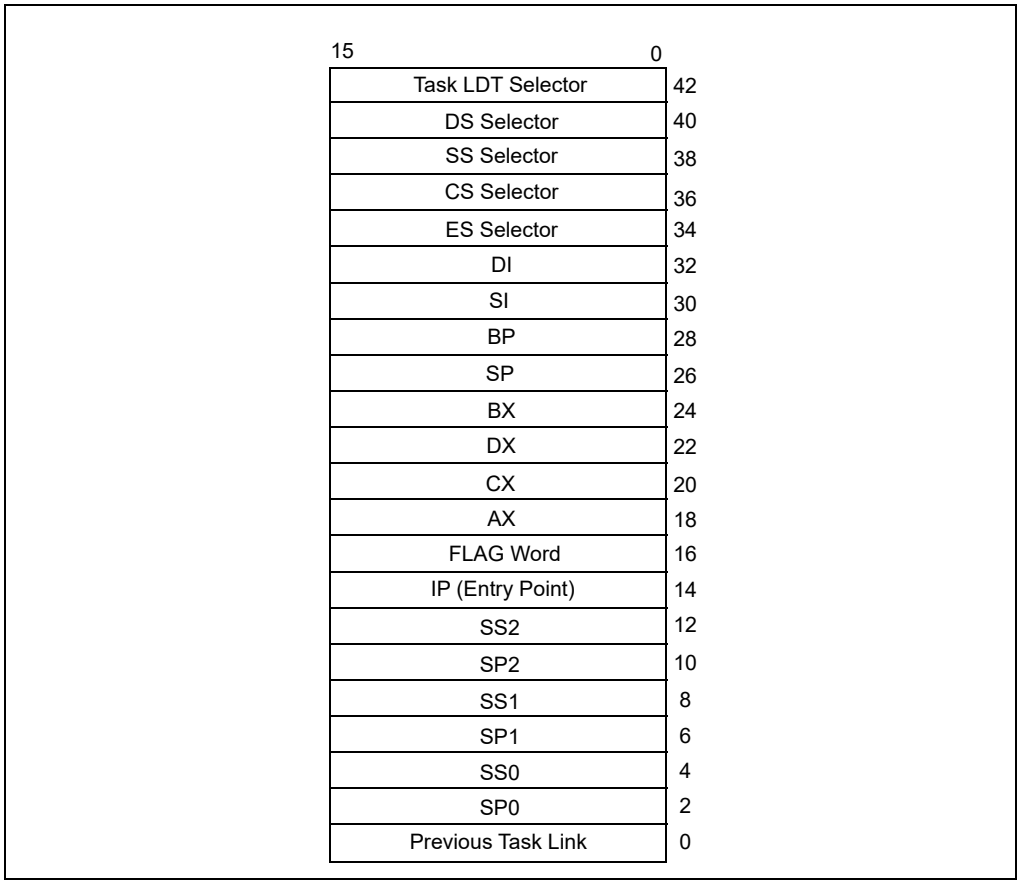


Figure 9-10. 16-Bit TSS Format

9.7 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 9-11 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.

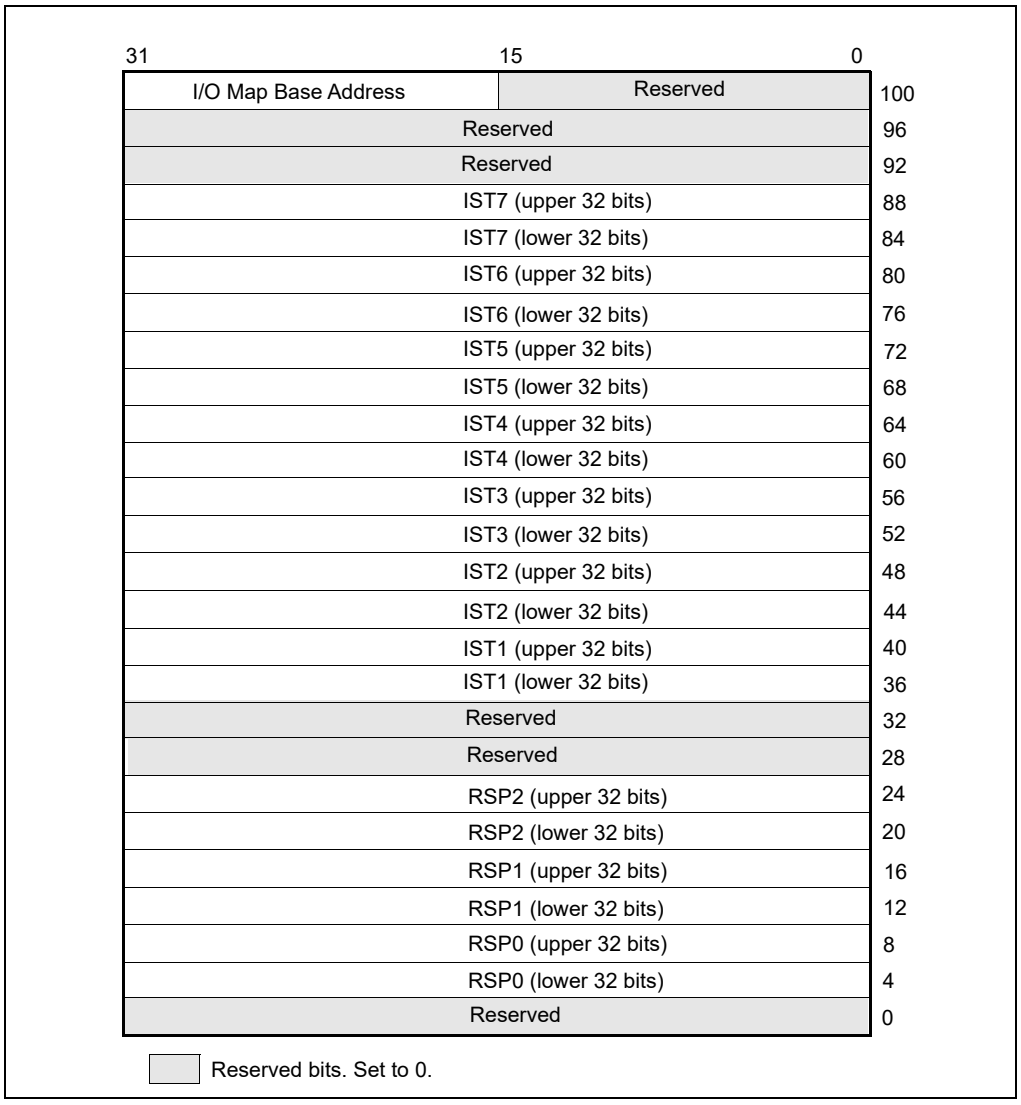


Figure 9-11. 64-Bit TSS Format

CHAPTER 10
MULTIPLE-PROCESSOR MANAGEMENT

The Intel 64 and IA-32 architectures provide mechanisms for managing and improving the performance of multiple processors connected to the same system bus. These include:

- Bus locking and/or cache coherency management for performing atomic operations on system memory.
- Serializing instructions.
- An advance programmable interrupt controller (APIC) located on the processor chip (see Chapter 12, "Advanced Programmable Interrupt Controller (APIC)"). This feature was introduced by the Pentium processor.
- A second-level cache (level 2, L2). For the Pentium 4, Intel Xeon, and P6 family processors, the L2 cache is included in the processor package and is tightly coupled to the processor. For the Pentium and Intel486 processors, pins are provided to support an external L2 cache.
- A third-level cache (level 3, L3). For Intel Xeon processors, the L3 cache is included in the processor package and is tightly coupled to the processor.
- Intel Hyper-Threading Technology. This extension to the Intel 64 and IA-32 architectures enables a single processor core to execute two or more threads concurrently (see Section 10.5, "Intel® Hyper-Threading Technology and Intel® Multi-Core Technology").

These mechanisms are particularly useful in symmetric-multiprocessing (SMP) systems. However, they can also be used when an Intel 64 or IA-32 processor and a special-purpose processor (such as a communications, graphics, or video processor) share the system bus.

These multiprocessing mechanisms have the following characteristics:

- To maintain system memory coherency — When two or more processors are attempting simultaneously to access the same address in system memory, some communication mechanism or memory access protocol must be available to promote data coherency and, in some instances, to allow one processor to temporarily lock a memory location.
- To maintain cache consistency — When one processor accesses data cached on another processor, it must not receive incorrect data. If it modifies data, all other processors that access that data must receive the modified data.
- To allow predictable ordering of writes to memory — In some circumstances, it is important that memory writes be observed externally in precisely the same order as programmed.
- To distribute interrupt handling among a group of processors — When several processors are operating in a system in parallel, it is useful to have a centralized mechanism for receiving interrupts and distributing them to available processors for servicing.
- To increase system performance by exploiting the multi-threaded and multi-process nature of contemporary operating systems and applications.

The caching mechanism and cache consistency of Intel 64 and IA-32 processors are discussed in Chapter 13. The APIC architecture is described in Chapter 12. Bus and memory locking, serializing instructions, memory ordering, and Intel Hyper-Threading Technology are discussed in the following sections.

10.1 LOCKED ATOMIC OPERATIONS

The 32-bit IA-32 processors support locked atomic operations on locations in system memory. These operations are typically used to manage shared data structures (such as semaphores, segment descriptors, system segments, or page tables) in which two or more processors may try simultaneously to modify the same field or flag. The processor uses three interdependent mechanisms for carrying out locked atomic operations:

- Guaranteed atomic operations.
- Bus locking, using the LOCK# signal and the LOCK instruction prefix.

- Cache coherency protocols that ensure that atomic operations can be carried out on cached data structures (cache lock); this mechanism is present in the Pentium 4, Intel Xeon, and P6 family processors.

These mechanisms are interdependent in the following ways. Certain basic memory transactions (such as reading or writing a byte in system memory) are always guaranteed to be handled atomically. That is, once started, the processor guarantees that the operation will be completed before another processor or bus agent is allowed access to the memory location. The processor also supports bus locking for performing selected memory operations (such as a read-modify-write operation in a shared area of memory) that typically need to be handled atomically, but are not automatically handled this way. Because frequently used memory locations are often cached in a processor's L1 or L2 caches, atomic operations can often be carried out inside a processor's caches without asserting the bus lock. Here the processor's cache coherency protocols ensure that other processors that are caching the same memory locations are managed properly while atomic operations are performed on cached memory locations.

NOTE

Where there are contested lock accesses, software may need to implement algorithms that ensure fair access to resources in order to prevent lock starvation. The hardware provides no resource that guarantees fairness to participating agents. It is the responsibility of software to manage the fairness of semaphores and exclusive locking functions.

The mechanisms for handling locked atomic operations have evolved with the complexity of IA-32 processors. More recent IA-32 processors (such as the Pentium 4, Intel Xeon, and P6 family processors) and Intel 64 provide a more refined locking mechanism than earlier processors. These mechanisms are described in the following sections.

10.1.1 Guaranteed Atomic Operations

The Intel486 processor (and newer processors since) guarantees that the following basic memory operations will always be carried out atomically:

- Reading or writing a byte.
- Reading or writing a word aligned on a 16-bit boundary.
- Reading or writing a doubleword aligned on a 32-bit boundary.

The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:

- Reading or writing a quadword aligned on a 64-bit boundary.
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus.

The P6 family processors (and newer processors since) guarantee that the following additional memory operation will always be carried out atomically:

- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line.

Processors that enumerate support for Intel® AVX (by setting the feature flag CPUID.01H:ECX.AVX[bit 28]) guarantee that the 16-byte memory operations performed by the following instructions will always be carried out atomically:

- MOVAPD, MOVAPS, and MOVDQA.
- VMOVAPD, VMOVAPS, and VMOVDQA when encoded with VEX.128.
- VMOVAPD, VMOVAPS, VMOVDQA32, and VMOVDQA64 when encoded with EVEX.128 and k0 (masking disabled).

(Note that these instructions require the linear addresses of their memory operands to be 16-byte aligned.)

Accesses to cacheable memory that are split across cache lines and page boundaries are not guaranteed to be atomic by the Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, and P6 family processors provide bus control signals that permit external memory subsystems to make split accesses atomic; however, nonaligned data accesses will seriously impact the performance of the processor and should be avoided.

Except as noted above, an x87 instruction or an SSE instruction that accesses data larger than a quadword may be implemented using multiple memory accesses. If such an instruction stores to memory, some of the accesses may complete (writing to memory) while another causes the operation to fault for architectural reasons (e.g., due an page-table entry that is marked "not present"). In this case, the effects of the completed accesses may be visible to software even though the overall instruction caused a fault. If TLB invalidation has been delayed (see Section 5.10.4.4), such page faults may occur even if all accesses are to the same page.

10.1.2 Bus Locking

Intel 64 and IA-32 processors provide a LOCK# signal that is asserted automatically during certain critical memory operations to lock the system bus or equivalent link. Assertion of this signal is called a **bus lock**. While this output signal is asserted, requests from other processors or bus agents for control of the bus are blocked. Software can specify other occasions when the LOCK semantics are to be followed by prepending the LOCK prefix to an instruction.

In the case of the Intel386, Intel486, and Pentium processors, explicitly locked instructions will result in the assertion of the LOCK# signal. It is the responsibility of the hardware designer to make the LOCK# signal available in system hardware to control memory accesses among processors.

For the P6 and more recent processor families, if the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted; instead, locking is only applied to the processor's caches (see Section 10.1.4, "Effects of a LOCK Operation on Internal Processor Caches"). These processors will assert a bus lock for a locked access in either of the following situations: (1) the access is to multiple cache lines (a **split lock**); or (2) the access uses a memory type other than WB (a **UC lock**)[1].

10.1.2.1 Automatic Locking

The operations on which the processor automatically follows the LOCK semantics are as follows:

- When executing an XCHG instruction that references memory.
- When switching to a task, the processor tests and sets the busy flag in the type field of the TSS descriptor. To ensure that two processors do not switch to the same task simultaneously, the processor follows the LOCK semantics while testing and setting this flag.
- When loading a segment descriptor, the processor sets the accessed flag in the segment descriptor if the flag is clear. During this operation, the processor follows the LOCK semantics so that the descriptor will not be modified by another processor while it is being updated. For this action to be effective, operating-system procedures that update descriptors should use the following steps:
  - Use a locked operation to modify the access-rights byte to indicate that the segment descriptor is not-present, and specify a value for the type field that indicates that the descriptor is being updated.
  - Update the fields of the segment descriptor. (This operation may require several memory accesses; therefore, locked operations cannot be used.)
  - Use a locked operation to modify the access-rights byte to indicate that the segment descriptor is valid and present.
  - The Intel386 processor always updates the accessed flag in the segment descriptor, whether it is clear or not. The Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors only update this flag if it is not already set.
- The processor uses locked cycles to set the accessed and dirty flag in paging-structure entries.
- After an interrupt request, an interrupt controller may use the data bus to send the interrupt's vector to the processor. The processor follows the LOCK semantics during this time to ensure that no other data appears on the data bus while the vector is being transmitted.

1. The term "UC lock" is used because the most common situation regards accesses to UC memory. Despite the name, locked accesses to WC, WP, and WT memory also cause bus locks.

10.1.2.2 Software Controlled Bus Locking

To explicitly force the LOCK semantics, software can use the LOCK prefix with the following instructions when they are used to modify a memory location. An invalid-opcode exception (#UD) is generated when the LOCK prefix is used with any other instruction or when no write operation is made to memory (that is, when the destination operand is in a register).

- The bit test and modify instructions (BTS, BTR, and BTC).
- The exchange instructions (XADD, CMPXCHG, CMPXCHG8B, and CMPXCHG16B).
- The LOCK prefix is automatically assumed for XCHG instruction.
- The following single-operand arithmetic and logical instructions: INC, DEC, NOT, and NEG.
- The following two-operand arithmetic and logical instructions: ADD, ADC, SUB, SBB, AND, OR, and XOR.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may be interpreted by the system as a lock for a larger memory area.

Software should access semaphores (shared memory used for signaling between multiple processors) using identical addresses and operand lengths. For example, if one processor accesses a semaphore using a word access, other processors should not access the semaphore using a byte access.

NOTE

Do not implement semaphores using the WC memory type. Do not perform non-temporal stores to a cache line containing a location used to implement a semaphore.

The integrity of a bus lock is not affected by the alignment of the memory field. The LOCK semantics are followed for as many bus cycles as necessary to update the entire operand. However, it is recommend that locked accesses be aligned on their natural boundaries for better system performance:

- Any boundary for an 8-bit access (locked or otherwise).
- 16-bit boundary for locked word accesses.
- 32-bit boundary for locked doubleword accesses.
- 64-bit boundary for locked quadword accesses.

Locked operations are atomic with respect to all other memory operations and all externally visible events. Only instruction fetch and page table accesses can pass locked instructions. Locked instructions can be used to synchronize data written by one processor and read by another processor.

For the P6 family processors, locked operations serialize all outstanding load and store operations (that is, wait for them to complete). This rule is also true for the Pentium 4 and Intel Xeon processors, with one exception. Load operations that reference weakly ordered memory types (such as the WC memory type) may not be serialized.

Locked instructions should not be used to ensure that data written can be fetched as instructions.

NOTE

The locked instructions for the current versions of the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors allow data written to be fetched as instructions. However, Intel recommends that developers who require the use of self-modifying code use a different synchronizing mechanism, described in the following sections.

10.1.2.3 Features to Disable Bus Locks

Because bus locks may adversely affect performance in certain situations, processors may support two features that system software can use to disable bus locking. These are called **split-lock disable** and **UC-lock disable**.

Support for split-lock disable is enumerated by IA32_CORE_CAPABILITIES[5].

Software enables split-lock disable by setting MSR_MEMORY_CTRL[29]. When this bit is set, a locked access to multiple cache lines causes a **bus-lock violation** that causes an alignment-check exception (#AC).[1] The locked access does not occur.

A processor enumerates support for UC-lock disable either by setting bit 4 of the IA32_CORE_CAPABILITIES MSR (MSR index CFH) or by enumerating CPUID.(EAX=07H, ECX=2):EDX[bit 6] as 1. The latter architectural form of enumeration (CPUID) is used beginning with processors based on Sierra Forest microarchitecture; earlier processors may use the older model-specific form (IA32_CORE_CAPABILITIES).

NOTE

No processor will both set IA32_CORE_CAPABILITIES[4] and enumerate CPUID.(EAX=07H, ECX=2):EDX[bit 6] as 1.

If a processor enumerates support for UC-lock disable (in either way), software can enable UC-lock disable by setting MSR_MEMORY_CTRL[28]. When this bit is set, a locked access using a memory type other than WB causes a bus-lock violation, leading to a fault. The locked access does not occur. The specific fault that occurs depends on how UC-lock disable is enumerated:

- If IA32_CORE_CAPABILITIES[4] is read as 1, the UC lock results in a general-protection exception (#GP) with a zero error code.
- If CPUID.(EAX=07H, ECX=2):EDX[bit 6] is enumerated as 1, the UC lock results in an #AC with an error code with value 4.[1]

UC-lock disable does not apply to locked accesses to physical addresses specified in a VMCS. Such accesses include updates to accessed and dirty flags for EPT and those to posted-interrupt descriptors.

UC-lock disable is not enabled if CR0.CD = 1 or if MSR_PRMRR_BASE_0[2:0] ≠ 6 (WB) when PRMRRs are enabled. If either of those conditions hold, the processor ignores the value of MSR_MEMORY_CTRL[28].

Note that the #AC(0) due to split-lock disable or alignment check is higher priority than a #GP(0) or #AC(4) due to UC-lock disable. If both features are enabled, a locked access to multiple cache lines causes #AC(0) regardless of the memory type(s) being accessed.

While MSR_MEMORY_CTRL is not an architectural MSR, the behavior described above is consistent across processor models that enumerate the support in IA32_CORE_CAPABILITIES or CPUID.

In addition to these features that disable bus locks, there are features that allow software to detect when a bus lock has occurred. See Section 19.3.1.6 for information about OS bus-lock detection and Section 27.2 for information about the VMM bus-lock detection.

10.1.3 Handling Self- and Cross-Modifying Code

The act of a processor writing data into a currently executing code segment with the intent of executing that data as code is called **self-modifying code**. IA-32 processors exhibit model-specific behavior when executing self-modified code, depending upon how far ahead of the current execution pointer the code has been modified.

As processor microarchitectures become more complex and start to speculatively execute code ahead of the retirement point (as in P6 and more recent processor families), the rules regarding which code should execute, pre- or post-modification, become blurred. To write self-modifying code and ensure that it is compliant with current and future versions of the IA-32 architectures, use one of the following coding options:

```
(* OPTION 1 *)
Store modified code (as data) into code segment;
Jump to new code or an intermediate location;
Execute new code;

(* OPTION 2 *)
Store modified code (as data) into code segment;
Execute a serializing instruction; (* For example, CPUID instruction *)
Execute new code;
```

1. Other alignment-check exceptions occur only if CR0.AM = 1, EFLAGS.AC = 1, and CPL = 3. The alignment-check exceptions resulting from split-lock disable may occur even if CR0.AM = 0, EFLAGS.AC = 0, or CPL < 3. For these bus-lock violations, the error code is null, meaning that bits 31:2 of the error code are clear; bit 0 (EXT) may be set normally.

1. Bit 0 (EXT) may be set normally. The error code would have value 5 if the EXT bit is set.

The use of one of these options is not required for programs intended to run on the Pentium or Intel486 processors, but are recommended to ensure compatibility with the P6 and more recent processor families.

Self-modifying code will execute at a lower level of performance than non-self-modifying or normal code. The degree of the performance deterioration will depend upon the frequency of modification and specific characteristics of the code.

The act of one processor writing data into the currently executing code segment of a second processor with the intent of having the second processor execute that data as code is called **cross-modifying code**. As with self-modifying code, IA-32 processors exhibit model-specific behavior when executing cross-modifying code, depending upon how far ahead of the executing processors current execution pointer the code has been modified.

To write cross-modifying code and ensure that it is compliant with current and future versions of the IA-32 architecture, the following processor synchronization algorithm must be implemented:

```
(* Action of Modifying Processor *)
Memory_Flag := 0; (* Set Memory_Flag to value other than 1 *)
Store modified code (as data) into code segment;
Memory_Flag := 1;

(* Action of Executing Processor *)
WHILE (Memory_Flag ≠ 1)
    Wait for code to update;
ELIHW;
Execute serializing instruction; (* For example, CPUID instruction *)
Begin executing modified code;
```

(The use of this option is not required for programs intended to run on the Intel486 processor, but is recommended to ensure compatibility with the Pentium 4, Intel Xeon, P6 family, and Pentium processors.)

Like self-modifying code, cross-modifying code will execute at a lower level of performance than non-cross-modifying (normal) code, depending upon the frequency of modification and specific characteristics of the code.

The restrictions on self-modifying code and cross-modifying code also apply to the Intel 64 architecture.

10.1.4 Effects of a LOCK Operation on Internal Processor Caches

For the Intel486 and Pentium processors, the LOCK# signal is always asserted on the bus during a LOCK operation, even if the area of memory being locked is cached in the processor.

For the P6 and more recent processor families, if the area of memory being locked during a LOCK operation is cached in the processor that is performing the LOCK operation as write-back memory and is completely contained in a cache line, the processor may not assert the LOCK# signal on the bus. Instead, it will modify the memory location internally and allow it's cache coherency mechanism to ensure that the operation is carried out atomically. This operation is called "cache locking." The cache coherency mechanism automatically prevents two or more processors that have cached the same area of memory from simultaneously modifying data in that area.

10.2 MEMORY ORDERING

The term **memory ordering** refers to the order in which the processor issues reads (loads) and writes (stores) through the system bus to system memory. The Intel 64 and IA-32 architectures support several memory-ordering models depending on the implementation of the architecture. For example, the Intel386 processor enforces **program ordering** (generally referred to as **strong ordering**), where reads and writes are issued on the system bus in the order they occur in the instruction stream under all circumstances.

To allow performance optimization of instruction execution, the IA-32 architecture allows departures from strong-ordering model called **processor ordering** in Pentium 4, Intel Xeon, and P6 family processors. These **processor-ordering** variations (called here the **memory-ordering model**) allow performance enhancing operations such as allowing reads to go ahead of buffered writes. The goal of any of these variations is to increase instruction execution speeds, while maintaining memory coherency, even in multiple-processor systems.

Section 10.2.1 and Section 10.2.2 describe the memory-ordering implemented by Intel486, Pentium, Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors. Section 10.2.3 gives examples illustrating the behavior of the memory-ordering model on IA-32 and Intel-64 processors. Section 10.2.4 considers the special treatment of stores for string operations and Section 10.2.5 discusses how memory-ordering behavior may be modified through the use of specific instructions.

10.2.1 Memory Ordering in the Intel® Pentium® and Intel486™ Processors

The Pentium and Intel486 processors follow the processor-ordered memory model; however, they operate as strongly-ordered processors under most circumstances. Reads and writes always appear in programmed order at the system bus—except for the following situation where processor ordering is exhibited. Read misses are permitted to go ahead of buffered writes on the system bus when all the buffered writes are cache hits and, therefore, are not directed to the same address being accessed by the read miss.

In the case of I/O operations, both reads and writes always appear in programmed order.

Software intended to operate correctly in processor-ordered processors (such as the Pentium 4, Intel Xeon, and P6 family processors) should not depend on the relatively strong ordering of the Pentium or Intel486 processors. Instead, it should ensure that accesses to shared variables that are intended to control concurrent execution among processors are explicitly required to obey program ordering through the use of appropriate locking or serializing operations (see Section 10.2.5, "Strengthening or Weakening the Memory-Ordering Model").

10.2.2 Memory Ordering in P6 and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as "write ordered with store-buffer forwarding." This model can be characterized as follows.

In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles (**Note** the memory-ordering principles for single-processor and multiple-processor systems are written from the perspective of software executing on the processor, where the term "processor" refers to a logical processor. For example, a physical processor supporting multiple cores and/or Intel Hyper-Threading Technology is treated as a multi-processor systems.):

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with other writes, with the following exceptions:
  - — streaming stores (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD); and
  - — string operations (see Section 10.2.4.1).
- No write to memory may be reordered with an execution of the CLFLUSH instruction; a write may be reordered with an execution of the CLFLUSHOPT instruction that flushes a cache line other than the one being written.[1] Executions of the CLFLUSH instruction are not reordered with each other. Executions of CLFLUSHOPT that access different cache lines may be reordered with each other. An execution of CLFLUSHOPT may be reordered with an execution of CLFLUSH that accesses a different cache line.
- Reads may be reordered with older writes to different locations but not with older writes to the same location.
- Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions.
- Reads cannot pass earlier LFENCE and MFENCE instructions.
- Writes and executions of CLFLUSH and CLFLUSHOPT cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.
- LFENCE instructions cannot pass earlier reads.
- SFENCE instructions cannot pass earlier writes or executions of CLFLUSH and CLFLUSHOPT.

1. Earlier versions of this manual specified that writes to memory may be reordered with executions of the CLFLUSH instruction. No processors implementing the CLFLUSH instruction allow such reordering.

- MFENCE instructions cannot pass earlier reads, writes, or executions of CLFLUSH and CLFLUSHOPT.

In a multiple-processor system, the following ordering principles apply:

- Individual processors use the same ordering principles as in a single-processor system.
- Writes by a single processor are observed in the same order by all processors.
- Writes from an individual processor are NOT ordered with respect to the writes from other processors.
- Memory ordering obeys causality (memory ordering respects transitive visibility).
- Any two stores are seen in a consistent order by processors other than those performing the stores
- Locked instructions have a total order.

See the example in Figure 10-1. Consider three processors in a system and each processor performs three writes, one to each of three defined locations (A, B, and C). Individually, the processors perform the writes in the same program order, but because of bus arbitration and other memory access mechanisms, the order that the three processors write the individual memory locations can differ each time the respective code sequences are executed on the processors. The final values in location A, B, and C would possibly vary on each execution of the write sequence.

The processor-ordering model described in this section is virtually identical to that used by the Pentium and Intel486 processors. The only enhancements in the Pentium 4, Intel Xeon, and P6 family processors are:

- Added support for speculative reads, while still adhering to the ordering principles above.
- Store-buffer forwarding, when a read passes a write to the same memory location.
- Out of order store from long string store and string move operations (see Section 10.2.4, “Fast-String Operation and Out-of-Order Stores,” below).

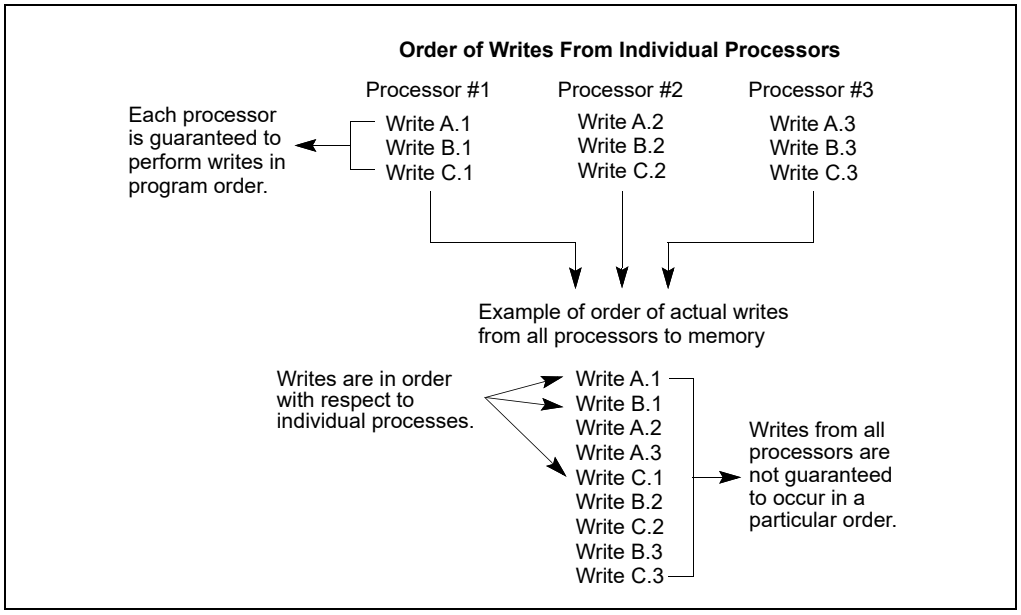


Figure 10-1. Example of Write Ordering in Multiple-Processor Systems

NOTE

In P6 processor family, store-buffer forwarding to reads of WC memory from streaming stores to the same address does not occur due to errata.

10.2.3 Examples Illustrating the Memory-Ordering Principles

This section provides a set of examples that illustrate the behavior of the memory-ordering principles introduced in Section 10.2.2. They are designed to give software writers an understanding of how memory ordering may affect the results of different sequences of instructions.

These examples are limited to accesses to memory regions defined as write-back cacheable (WB). (Section 10.2.3.1 describes other limitations on the generality of the examples.) The reader should understand that they describe only software-visible behavior. A logical processor may reorder two accesses even if one of examples indicates that they may not be reordered. Such an example states only that software cannot detect that such a reordering occurred. Similarly, a logical processor may execute a memory access more than once as long as the behavior visible to software is consistent with a single execution of the memory access.

10.2.3.1 Assumptions, Terminology, and Notation

As noted above, the examples in this section are limited to accesses to memory regions defined as write-back cacheable (WB). They apply only to ordinary loads stores and to locked read-modify-write instructions. They do not necessarily apply to any of the following: out-of-order stores for string instructions (see Section 10.2.4); accesses with a non-temporal hint; reads from memory by the processor as part of address translation (e.g., page walks); and updates to segmentation and paging structures by the processor (e.g., to update "accessed" bits).

The principles underlying the examples in this section apply to individual memory accesses and to locked read-modify-write instructions. The Intel-64 memory-ordering model guarantees that, for each of the following memory-access instructions, the constituent memory operation appears to execute as a single memory access:

- Instructions that read or write a single byte.
- Instructions that read or write a word (2 bytes) whose address is aligned on a 2 byte boundary.
- Instructions that read or write a doubleword (4 bytes) whose address is aligned on a 4 byte boundary.
- Instructions that read or write a quadword (8 bytes) whose address is aligned on an 8 byte boundary.

Any locked instruction (either the XCHG instruction or another read-modify-write instruction with a LOCK prefix) appears to execute as an indivisible and uninterruptible sequence of load(s) followed by store(s) regardless of alignment.

Other instructions may be implemented with multiple memory accesses. From a memory-ordering point of view, there are no guarantees regarding the relative order in which the constituent memory accesses are made. There is also no guarantee that the constituent operations of a store are executed in the same order as the constituent operations of a load.

Section 10.2.3.2 through Section 10.2.3.7 give examples using the MOV instruction. The principles that underlie these examples apply to load and store accesses in general and to other instructions that load from or store to memory. Section 10.2.3.8 and Section 10.2.3.9 give examples using the XCHG instruction. The principles that underlie these examples apply to other locked read-modify-write instructions.

This section uses the term "processor" is to refer to a logical processor. The examples are written using Intel-64 assembly-language syntax and use the following notational conventions:

- Arguments beginning with an "r", such as r1 or r2 refer to registers (e.g., EAX) visible only to the processor being considered.
- Memory locations are denoted with x, y, z.
- Stores are written as *mov [_x], val*, which implies that *val* is being stored into the memory location *x*.
- Loads are written as *mov r, [_x]*, which implies that the contents of the memory location *x* are being loaded into the register *r*.

As noted earlier, the examples refer only to software visible behavior. When the succeeding sections make statement such as "the two stores are reordered," the implication is only that "the two stores appear to be reordered from the point of view of software."

10.2.3.2 Neither Loads Nor Stores Are Reordered with Like Operations

The Intel-64 memory-ordering model allows neither loads nor stores to be reordered with the same kind of operation. That is, it ensures that loads are seen in program order and that stores are seen in program order. This is illustrated by the following example:

Example 10-1. Stores Are Not Reordered with Other Stores

Processor 0	Processor 1
mov [_x], 1 mov [_y], 1	mov r1, [_y] mov r2, [_x]
Initially x = y = 0 r1 = 1 and r2 = 0 is not allowed	

The disallowed return values could be exhibited only if processor 0's two stores are reordered (with the two loads occurring between them) or if processor 1's two loads are reordered (with the two stores occurring between them).

If r1 = 1, the store to y occurs before the load from y. Because the Intel-64 memory-ordering model does not allow stores to be reordered, the earlier store to x occurs before the load from y. Because the Intel-64 memory-ordering model does not allow loads to be reordered, the store to x also occurs before the later load from x. This r2 = 1.

10.2.3.3 Stores Are Not Reordered With Earlier Loads

The Intel-64 memory-ordering model ensures that a store by a processor may not occur before a previous load by the same processor. This is illustrated in Example 10-2.

Example 10-2. Stores Are Not Reordered with Older Loads

Processor 0	Processor 1
mov r1, [_x]	mov r2, [_y]
mov [_y], 1	mov [_x], 1
Initially x = y = 0	
r1 = 1 and r2 = 1 is not allowed	

Assume r1 = 1.

- Because r1 = 1, processor 1's store to x occurs before processor 0's load from x.
- Because the Intel-64 memory-ordering model prevents each store from being reordered with the earlier load by the same processor, processor 1's load from y occurs before its store to x.
- Similarly, processor 0's load from x occurs before its store to y.
- Thus, processor 1's load from y occurs before processor 0's store to y, implying r2 = 0.

10.2.3.4 Loads May Be Reordered with Earlier Stores to Different Locations

The Intel-64 memory-ordering model allows a load to be reordered with an earlier store to a different location. However, loads are not reordered with stores to the same location.

The fact that a load may be reordered with an earlier store to a different location is illustrated by the following example:

Example 10-3. Loads May be Reordered with Older Stores

Processor 0	Processor 1
mov [_x], 1	mov [_y], 1
mov r1, [_y]	mov r2, [_x]
Initially x = y = 0	
r1 = 0 and r2 = 0 is allowed	

At each processor, the load and the store are to different locations and hence may be reordered. Any interleaving of the operations is thus allowed. One such interleaving has the two loads occurring before the two stores. This would result in each load returning value 0.

The fact that a load may not be reordered with an earlier store to the same location is illustrated by the following example:

Example 10-4. Loads Are not Reordered with Older Stores to the Same Location

Processor 0
mov [_x], 1
mov r1, [_x]
Initially x = 0
r1 = 0 is not allowed

The Intel-64 memory-ordering model does not allow the load to be reordered with the earlier store because the accesses are to the same location. Therefore, r1 = 1 must hold.

10.2.3.5 Intra-Processor Forwarding Is Allowed

The memory-ordering model allows concurrent stores by two processors to be seen in different orders by those two processors; specifically, each processor may perceive its own store occurring before that of the other. This is illustrated by the following example:

Example 10-5. Intra-Processor Forwarding is Allowed

Processor 0	Processor 1
mov [_x], 1 mov r1, [_x] mov r2, [_y]	mov [_y], 1 mov r3, [_y] mov r4, [_x]
Initially x = y = 0 r2 = 0 and r4 = 0 is allowed	

The memory-ordering model imposes no constraints on the order in which the two stores appear to execute by the two processors. This fact allows processor 0 to see its store before seeing processor 1's, while processor 1 sees its store before seeing processor 0's. (Each processor is self consistent.) This allows r2 = 0 and r4 = 0.

In practice, the reordering in this example can arise as a result of store-buffer forwarding. While a store is temporarily held in a processor's store buffer, it can satisfy the processor's own loads but is not visible to (and cannot satisfy) loads by other processors.

10.2.3.6 Stores Are Transitively Visible

The memory-ordering model ensures transitive visibility of stores; stores that are causally related appear to all processors to occur in an order consistent with the causality relation. This is illustrated by the following example:

Example 10-6. Stores Are Transitively Visible

Processor 0	Processor 1	Processor 2
mov [_x], 1	mov r1, [_x] mov [_y], 1	 mov r2, [_y] mov r3, [_x]
Initially x = y = 0 r1 = 1, r2 = 1, r3 = 0 is not allowed		

Assume that r1 = 1 and r2 = 1.

- Because r1 = 1, processor 0’s store occurs before processor 1’s load.
- Because the memory-ordering model prevents a store from being reordered with an earlier load (see Section 10.2.3.3), processor 1’s load occurs before its store. Thus, processor 0’s store causally precedes processor 1’s store.
- Because processor 0’s store causally precedes processor 1’s store, the memory-ordering model ensures that processor 0’s store appears to occur before processor 1’s store from the point of view of all processors.
- Because r2 = 1, processor 1’s store occurs before processor 2’s load.
- Because the Intel-64 memory-ordering model prevents loads from being reordered (see Section 10.2.3.2), processor 2’s load occur in order.
- The above items imply that processor 0’s store to x occurs before processor 2’s load from x. This implies that r3 = 1.

10.2.3.7 Stores Are Seen in a Consistent Order by Other Processors

As noted in Section 10.2.3.5, the memory-ordering model allows stores by two processors to be seen in different orders by those two processors. However, any two stores must appear to execute in the same order to all processors other than those performing the stores. This is illustrated by the following example:

Example 10-7. Stores Are Seen in a Consistent Order by Other Processors

Processor 0	Processor 1	Processor 2	Processor 3
mov [_x], 1	mov [_y], 1	mov r1, [_x] mov r2, [_y]	mov r3, [_y] mov r4, [_x]
Initially x = y =0 r1 = 1, r2 = 0, r3 = 1, r4 = 0 is not allowed			

By the principles discussed in Section 10.2.3.2:

- Processor 2's first and second load cannot be reordered.
- Processor 3's first and second load cannot be reordered.
- If r1 = 1 and r2 = 0, processor 0's store appears to precede processor 1's store with respect to processor 2.
- Similarly, r3 = 1 and r4 = 0 imply that processor 1's store appears to precede processor 0's store with respect to processor 1.

Because the memory-ordering model ensures that any two stores appear to execute in the same order to all processors (other than those performing the stores), this set of return values is not allowed.

10.2.3.8 Locked Instructions Have a Total Order

The memory-ordering model ensures that all processors agree on a single execution order of all locked instructions, including those that are larger than 8 bytes or are not naturally aligned. This is illustrated by the following example:

Example 10-8. Locked Instructions Have a Total Order

Processor 0	Processor 1	Processor 2	Processor 3

xchg [_x], r1	xchg [_y], r2	
		mov r3, [_x]
		mov r4, [_y]
			
			mov r5, [_y]
			mov r6, [_x]

Initially r1 = r2 = 1, x = y = 0 r3 = 1, r4 = 0, r5 = 1, r6 = 0 is not allowed			

Processor 2 and processor 3 must agree on the order of the two executions of XCHG. Without loss of generality, suppose that processor 0's XCHG occurs first.

- If r5 = 1, processor 1's XCHG into y occurs before processor 3's load from y.
- Because the Intel-64 memory-ordering model prevents loads from being reordered (see Section 10.2.3.2), processor 3's loads occur in order and, therefore, processor 1's XCHG occurs before processor 3's load from x.
- Since processor 0's XCHG into x occurs before processor 1's XCHG (by assumption), it occurs before processor 3's load from x. Thus, r6 = 1.

A similar argument (referring instead to processor 2's loads) applies if processor 1's XCHG occurs before processor 0's XCHG.

10.2.3.9 Loads and Stores Are Not Reordered with Locked Instructions

The memory-ordering model prevents loads and stores from being reordered with locked instructions that execute earlier or later. The examples in this section illustrate only cases in which a locked instruction is executed before a

load or a store. The reader should note that reordering is prevented also if the locked instruction is executed after a load or a store.

The first example illustrates that loads may not be reordered with earlier locked instructions:

Example 10-9. Loads Are not Reordered with Locks

Processor 0	Processor 1
xchg [_x], r1 mov r2, [_y]	xchg [_y], r3 mov r4, [_x]
Initially x = y = 0, r1 = r3 = 1 r2 = 0 and r4 = 0 is not allowed	

As explained in Section 10.2.3.8, there is a total order of the executions of locked instructions. Without loss of generality, suppose that processor 0's XCHG occurs first.

Because the Intel-64 memory-ordering model prevents processor 1's load from being reordered with its earlier XCHG, processor 0's XCHG occurs before processor 1's load. This implies r4 = 1.

A similar argument (referring instead to processor 2's accesses) applies if processor 1's XCHG occurs before processor 0's XCHG.

The second example illustrates that a store may not be reordered with an earlier locked instruction:

Example 10-10. Stores Are not Reordered with Locks

Processor 0	Processor 1
xchg [_x], r1 mov [_y], 1	mov r2, [_y] mov r3, [_x]
Initially x = y = 0, r1 = 1 r2 = 1 and r3 = 0 is not allowed	

Assume r2 = 1.

- Because r2 = 1, processor 0's store to y occurs before processor 1's load from y.
- Because the memory-ordering model prevents a store from being reordered with an earlier locked instruction, processor 0's XCHG into x occurs before its store to y. Thus, processor 0's XCHG into x occurs before processor 1's load from y.
- Because the memory-ordering model prevents loads from being reordered (see Section 10.2.3.2), processor 1's loads occur in order and, therefore, processor 1's XCHG into x occurs before processor 1's load from x. Thus, r3 = 1.

10.2.4 Fast-String Operation and Out-of-Order Stores

Section 7.3.9.3 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, described an optimization of repeated string operations called **fast-string operation**.

As explained in that section, the stores produced by fast-string operation may appear to execute out of order. Software dependent upon sequential store ordering should not use string operations for the entire data structure to be stored. Data and semaphores should be separated. Order-dependent code should write to a discrete semaphore variable after any string operations to allow correctly ordered data to be seen by all processors. Atomicity of load and store operations is guaranteed only for native data elements of the string with native data size, and only if they are included in a single cache line.

Section 10.2.4.1 and Section 10.2.4.2 provide further explain and examples.

10.2.4.1 Memory-Ordering Model for String Operations on Write-Back (WB) Memory

This section deals with the memory-ordering model for string operations on write-back (WB) memory for the Intel 64 architecture.

The memory-ordering model respects the follow principles:

1. Stores within a single string operation may be executed out of order.
2. Stores from separate string operations (for example, stores from consecutive string operations) do not execute out of order. All the stores from an earlier string operation will complete before any store from a later string operation.
3. String operations are not reordered with other store operations.

Fast string operations (e.g., string operations initiated with the MOVS/STOS instructions and the REP prefix) may be interrupted by exceptions or interrupts. The interrupts are precise but may be delayed - for example, the interruptions may be taken at cache line boundaries, after every few iterations of the loop, or after operating on every few bytes. Different implementations may choose different options, or may even choose not to delay interrupt handling, so software should not rely on the delay. When the interrupt/trap handler is reached, the source/destination registers point to the next string element to be operated on, while the EIP stored in the stack points to the string instruction, and the ECX register has the value it held following the last successful iteration. The return from that trap/interrupt handler should cause the string instruction to be resumed from the point where it was interrupted.

The string operation memory-ordering principles, (item 2 and 3 above) should be interpreted by taking the incorruptibility of fast string operations into account. For example, if a fast string operation gets interrupted after k iterations, then stores performed by the interrupt handler will become visible after the fast string stores from iteration 0 to k, and before the fast string stores from the (k+1)th iteration onward.

Stores within a single string operation may execute out of order (item 1 above) only if fast string operation is enabled. Fast string operations are enabled/disabled through the IA32_MISC_ENABLE model specific register.

10.2.4.2 Examples Illustrating Memory-Ordering Principles for String Operations

The following examples uses the same notation and convention as described in Section 10.2.3.1.

In Example 10-11, processor 0 does one round of (128 iterations) doubleword string store operation via rep:stosd, writing the value 1 (value in EAX) into a block of 512 bytes from location _x (kept in ES:EDI) in ascending order. Since each operation stores a doubleword (4 bytes), the operation is repeated 128 times (value in ECX). The block of memory initially contained 0. Processor 1 is reading two memory locations that are part of the memory block being updated by processor 0, i.e, reading locations in the range _x to (_x+511).

Example 10-11. Stores Within a String Operation May be Reordered

Processor 0	Processor 1
rep:stosd [_x]	mov r1, [_z] mov r2, [_y]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially [_x] to 511[_x]= 0, _x <= _y < _z < _x+512 r1 = 1 and r2 = 0 is allowed	

It is possible for processor 1 to perceive that the repeated string stores in processor 0 are happening out of order. Assume that fast string operations are enabled on processor 0.

In Example 10-12, processor 0 does two separate rounds of rep stosd operation of 128 doubleword stores, writing the value 1 (value in EAX) into the first block of 512 bytes from location _x (kept in ES:EDI) in ascending order. It then writes 1 into a second block of memory from (_x+512) to (_x+1023). All of the memory locations initially contain 0. The block of memory initially contained 0. Processor 1 performs two load operations from the two blocks of memory.

Example 10-12. Stores Across String Operations Are not Reordered

Processor 0	Processor 1
rep:stosd [_x] mov ecx, $128 rep:stosd 512[_x]	 mov r1, [_z] mov r2, [_y]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially [_x] to 1023[_x]= 0, _x <= _y < _x+512 < _z < _x+1024 r1 = 1 and r2 = 0 is not allowed	

It is not possible in the above example for processor 1 to perceive any of the stores from the later string operation (to the second 512 block) in processor 0 before seeing the stores from the earlier string operation to the first 512 block.

The above example assumes that writes to the second block (_x+512 to _x+1023) does not get executed while processor 0's string operation to the first block has been interrupted. If the string operation to the first block by processor 0 is interrupted, and a write to the second memory block is executed by the interrupt handler, then that change in the second memory block will be visible before the string operation to the first memory block resumes.

In Example 10-13, processor 0 does one round of (128 iterations) doubleword string store operation via rep:stosd, writing the value 1 (value in EAX) into a block of 512 bytes from location _x (kept in ES:EDI) in ascending order. It then writes to a second memory location outside the memory block of the previous string operation. Processor 1 performs two read operations, the first read is from an address outside the 512-byte block but to be updated by processor 0, the second ready is from inside the block of memory of string operation.

Example 10-13. String Operations Are not Reordered with later Stores

Processor 0	Processor 1
rep:stosd [_x] mov [_z], $1	mov r1, [_z] mov r2, [_y]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially [_y] = [_z] = 0, [_x] to 511[_x]= 0, _x <= _y < _x+512, _z is a separate memory location r1 = 1 and r2 = 0 is not allowed	

Processor 1 cannot perceive the later store by processor 0 until it sees all the stores from the string operation. Example 10-13 assumes that processor 0's store to [_z] is not executed while the string operation has been interrupted. If the string operation is interrupted and the store to [_z] by processor 0 is executed by the interrupt handler, then changes to [_z] will become visible before the string operation resumes.

Example 10-14 illustrates the visibility principle when a string operation is interrupted.

Example 10-14. Interrupted String Operation

Processor 0	Processor 1
rep:stosd [_x] // interrupted before es:edi reach _y mov [_z], $1 // interrupt handler	mov r1, [_z] mov r2, [_y]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially [_y] = [_z] = 0, [_x] to 511[_x]= 0, _x <= _y < _x+512, _z is a separate memory location r1 = 1 and r2 = 0 is allowed	

In Example 10-14, processor 0 started a string operation to write to a memory block of 512 bytes starting at address _x. Processor 0 got interrupted after k iterations of store operations. The address _y has not yet been updated by processor 0 when processor 0 got interrupted. The interrupt handler that took control on processor 0 writes to the address _z. Processor 1 may see the store to _z from the interrupt handler, before seeing the remaining stores to the 512-byte memory block that are executed when the string operation resumes.

Example 10-15 illustrates the ordering of string operations with earlier stores. No store from a string operation can be visible before all prior stores are visible.

Example 10-15. String Operations Are not Reordered with Earlier Stores

Processor 0	Processor 1
mov [_z], $1 rep:stosd [_x]	mov r1, [_y] mov r2, [_z]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially [_y] = [_z] = 0, [_x] to 511[_x]= 0, _x <= _y < _x+512, _z is a separate memory location r1 = 1 and r2 = 0 is not allowed	

10.2.5 Strengthening or Weakening the Memory-Ordering Model

The Intel 64 and IA-32 architectures provide several mechanisms for strengthening or weakening the memory-ordering model to handle special programming situations. These mechanisms include:

- The I/O instructions, locked instructions, the LOCK prefix, and serializing instructions force stronger ordering on the processor.
- The SFENCE instruction (introduced to the IA-32 architecture in the Pentium III processor) and the LFENCE and MFENCE instructions (introduced in the Pentium 4 processor) provide memory-ordering and serialization capabilities for specific types of memory operations.
- The memory type range registers (MTRRs) can be used to strengthen or weaken memory ordering for specific area of physical memory (see Section 13.11, "Memory Type Range Registers (MTRRs)"). MTRRs are available only in the Pentium 4, Intel Xeon, and P6 family processors.
- The page attribute table (PAT) can be used to strengthen memory ordering for a specific page or group of pages (see Section 13.12, "Page Attribute Table (PAT)"). The PAT is available only in the Pentium 4, Intel Xeon, and Pentium III processors.

These mechanisms can be used as follows:

Memory mapped devices and other I/O devices on the bus are often sensitive to the order of writes to their I/O buffers. I/O instructions can be used to (the IN and OUT instructions) impose strong write ordering on such accesses as follows. Prior to executing an I/O instruction, the processor waits for all previous instructions in the program to complete and for all buffered writes to drain to memory. Only instruction fetch and page tables walks can pass I/O instructions. Execution of subsequent instructions do not begin until the processor determines that the I/O instruction has been completed.

Synchronization mechanisms in multiple-processor systems may depend upon a strong memory-ordering model. Here, a program can use a locked instruction such as the XCHG instruction or the LOCK prefix to ensure that a read-modify-write operation on memory is carried out atomically. Locked instructions typically operate like I/O instructions in that they wait for all previous memory accesses to complete and for all buffered writes to drain to memory (see Section 10.1.2, "Bus Locking"). Unlike I/O operations, locked instructions do not wait for all previous instructions to complete execution.

Program synchronization can also be carried out with serializing instructions (see Section 10.3). These instructions are typically used at critical procedure or task boundaries to force completion of all previous instructions before a jump to a new section of code or a context switch occurs. Like the I/O instructions, the processor waits until all previous instructions have been completed and all buffered writes have been drained to memory before executing the serializing instruction.

The SFENCE, LFENCE, and MFENCE instructions provide a performance-efficient way of ensuring load and store memory ordering between routines that produce weakly-ordered results and routines that consume that data. The functions of these instructions are as follows:

- **SFENCE** — Serializes all store (write) operations that occurred prior to the SFENCE instruction in the program instruction stream, but does not affect load operations.
- **LFENCE** — Serializes all load (read) operations that occurred prior to the LFENCE instruction in the program instruction stream, but does not affect store operations.[1]
- **MFENCE** — Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

Note that the SFENCE, LFENCE, and MFENCE instructions provide a more efficient method of controlling memory ordering than the CPUID instruction.

The MTRRs were introduced in the P6 family processors to define the cache characteristics for specified areas of physical memory. The following are two examples of how memory types set up with MTRRs can be used strengthen or weaken memory ordering for the Pentium 4, Intel Xeon, and P6 family processors:

- The strong uncached (UC) memory type forces a strong-ordering model on memory accesses. Here, all reads and writes to the UC memory region appear on the bus and out-of-order or speculative accesses are not performed. This memory type can be applied to an address range dedicated to memory mapped I/O devices to force strong memory ordering.
- For areas of memory where weak ordering is acceptable, the write back (WB) memory type can be chosen. Here, reads can be performed speculatively and writes can be buffered and combined. For this type of memory, cache locking is performed on atomic (locked) operations that do not split across cache lines, which helps to reduce the performance penalty associated with the use of the typical synchronization instructions, such as XCHG, that lock the bus during the entire read-modify-write operation. With the WB memory type, the XCHG instruction locks the cache instead of the bus if the memory access is contained within a cache line.

The PAT was introduced in the Pentium III processor to enhance the caching characteristics that can be assigned to pages or groups of pages. The PAT mechanism typically used to strengthen caching characteristics at the page level with respect to the caching characteristics established by the MTRRs. Table 13-7 shows the interaction of the PAT with the MTRRs.

Intel recommends that software written to run on Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors assume the processor-ordering model or a weaker memory-ordering model. The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors do not implement a strong memory-ordering model, except when using the UC memory type. Despite the fact that Pentium 4, Intel Xeon, and P6 family processors support processor ordering, Intel does not guarantee that future processors will support this model. To make software portable to future processors, it is recommended that operating systems provide critical region and resource control constructs and API's (application program interfaces) based on I/O, locking, and/or serializing instructions be used to synchronize access to shared areas of memory in multiple-processor systems. Also, software should not depend on processor ordering in situations where the system hardware does not support this memory-ordering model.

10.3 SERIALIZING INSTRUCTIONS

The Intel 64 and IA-32 architectures define several **serializing instructions**. These instructions force the processor to complete all modifications to flags, registers, and memory by previous instructions and to drain all buffered writes to memory before the next instruction is fetched and executed. For example, when a MOV to control register instruction is used to load a new value into control register CR0 to enable protected mode, the processor must perform a serializing operation before it enters protected mode. This serializing operation ensures that all

1. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. As a result, an instruction that loads from memory and that precedes an LFENCE receives data from memory prior to completion of the LFENCE. An LFENCE that follows an instruction that stores to memory might complete before the data being stored have become globally visible. Instructions following an LFENCE may be fetched from memory before the LFENCE, but they will not execute until the LFENCE completes.

operations that were started while the processor was in real-address mode are completed before the switch to protected mode is made.

The concept of serializing instructions was introduced into the IA-32 architecture with the Pentium processor to support parallel instruction execution. Serializing instructions have no meaning for the Intel486 and earlier processors that do not implement parallel instruction execution.

It is important to note that executing of serializing instructions on P6 and more recent processor families constrain speculative execution because the results of speculatively executed instructions are discarded. The following instructions are serializing instructions:

- **Privileged serializing instructions** — INVD, INVEPT, INVLPG, INVVPID, LGDT, LIDT, LLDT, LTR, MOV (to control register, with the exception of MOV CR8[1]), MOV (to debug register), WBINVD, and WRMSR[2].
- **Non-privileged serializing instructions** — CPUID, IRET, RSM, and SERIALIZE.

When the processor serializes instruction execution, it ensures that all pending memory transactions are completed (including writes stored in its store buffer) before it executes the next _instruction. Nothing can pass a serializing instruction and a serializing instruction cannot pass any other instruction (read, write, instruction fetch, or I/O). For example, CPUID can be executed at any privilege level to serialize instruction execution with no effect on program flow, except that the EAX, EBX, ECX, and EDX registers are modified.

The following instructions are memory-ordering instructions, not serializing instructions. These drain the data memory subsystem. They do not serialize the instruction execution stream:[3]

- **Non-privileged memory-ordering instructions** — SFENCE, LFENCE, and MFENCE.

The SFENCE, LFENCE, and MFENCE instructions provide more granularity in controlling the serialization of memory loads and stores (see Section 10.2.5, "Strengthening or Weakening the Memory-Ordering Model").

The following additional information is worth noting regarding serializing instructions:

- The processor does not write back the contents of modified data in its data cache to external memory when it serializes instruction execution. Software can force modified data to be written back by executing the WBINVD instruction, which is a serializing instruction. The amount of time or cycles for WBINVD to complete will vary due to the size of different cache hierarchies and other factors. As a consequence, the use of the WBINVD instruction can have an impact on interrupt/event response time.
- When an instruction is executed that enables or disables paging (that is, changes the PG flag in control register CR0), the instruction should be followed by a jump instruction. The target instruction of the jump instruction is fetched with the new setting of the PG flag (that is, paging is enabled or disabled), but the jump instruction itself is fetched with the previous setting. The Pentium 4, Intel Xeon, and P6 family processors do not require the jump operation following the move to register CR0 (because any use of the MOV instruction in a Pentium 4, Intel Xeon, or P6 family processor to write to CR0 is completely serializing). However, to maintain backwards and forward compatibility with code written to run on other IA-32 processors, it is recommended that the jump operation be performed.
- Whenever an instruction is executed to change the contents of CR3 while paging is enabled, the next instruction is fetched using the translation tables that correspond to the new value of CR3. Therefore the next instruction and the sequentially following instructions should have a mapping based upon the new value of CR3. (Global entries in the TLBs are not invalidated, see Section 5.10.4, "Invalidation of TLBs and Paging-Structure Caches.")
- The Pentium processor and more recent processor families use branch-prediction techniques to improve performance by prefetching the destination of a branch instruction before the branch instruction is executed. Consequently, instruction execution is not deterministically serialized when a branch instruction is executed.

---

1. MOV CR8 is not defined architecturally as a serializing instruction.
2. An execution of WRMSR to any non-serializing MSR is not serializing. Non-serializing MSRs include the following: IA32_SPEC_CTRL MSR (MSR index 48H), IA32_PRED_CMD MSR (MSR index 49H), IA32_TSX_CTRL MSR (MSR index 122H), IA32_TSC_DEADLINE MSR (MSR index 6E0H), IA32_PKRS MSR (MSR index 6E1H), IA32_HWP_REQUEST MSR (MSR index 774H), or any of the x2APIC MSRs (MSR indices 802H to 83FH).
3. LFENCE does provide some guarantees on instruction ordering. It does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes.

10.4 MULTIPLE-PROCESSOR (MP) INITIALIZATION

The IA-32 architecture (beginning with the P6 family processors) defines a multiple-processor (MP) initialization protocol called the Multiprocessor Specification Version 1.4. This specification defines the boot protocol to be used by IA-32 processors in multiple-processor systems. (Here, **multiple processors** is defined as two or more processors.) The MP initialization protocol has the following important features:

- It supports controlled booting of multiple processors without requiring dedicated system hardware.
- It allows hardware to initiate the booting of a system without the need for a dedicated signal or a predefined boot processor.
- It allows all IA-32 processors to be booted in the same manner, including those supporting Intel Hyper-Threading Technology.
- The MP initialization protocol also applies to MP systems using Intel 64 processors.

The mechanism for carrying out the MP initialization protocol differs depending on the Intel processor generations. The following bullets summarizes the evolution of the changes:

- **For P6 family or older processors supporting MP operations**— The selection of the BSP and APs (see Section 10.4.1, "BSP and AP Processors") is handled through arbitration on the APIC bus, using BIPI and FIPI messages. These processor generations have CPUID signatures of (family=06H, extended_model=0, model<=0DH), or family <06H. See Section 10.11.1, "Overview of the MP Initialization Process for P6 Family Processors," for a complete discussion of MP initialization for P6 family processors.
- **Early generations of IA processors with family 0FH** — The selection of the BSP and APs (see Section 10.4.1, "BSP and AP Processors") is handled through arbitration on the system bus, using BIPI and FIPI messages (see Section 10.4.3, "MP Initialization Protocol Algorithm for MP Systems"). These processor generations have CPUID signatures of family=0FH, model=0H, stepping<=09H.
- **Later generations of IA processors with family 0FH, and IA processors with system bus** — The selection of the BSP and APs is handled through a special system bus cycle, without using BIPI and FIPI message arbitration (see Section 10.4.3, "MP Initialization Protocol Algorithm for MP Systems"). These processor generations have CPUID signatures of family=0FH with (model=0H, stepping>=0AH) or (model >0, all steppings); or family=06H, extended_model=0, model>=0EH.
- **All other modern IA processor generations supporting MP operations**— The selection of the BSP and APs in the system is handled by platform-specific arrangement of the combination of hardware, BIOS, and/or configuration input options. The basis of the selection mechanism is similar to those of the Later generations of family 0FH and other Intel processor using system bus (see Section 10.4.3, "MP Initialization Protocol Algorithm for MP Systems"). These processor generations have CPUID signatures of family=06H, extended_-model>0.

The family, model, and stepping ID for a processor is given in the EAX register when the CPUID instruction is executed with a value of 1 in the EAX register.

10.4.1 BSP and AP Processors

The MP initialization protocol defines two classes of processors: the bootstrap processor (BSP) and the application processors (APs). Following a power-up or RESET of an MP system, system hardware dynamically selects one of the processors on the system bus as the BSP. The remaining processors are designated as APs.

As part of the BSP selection mechanism, the BSP flag is set in the IA32_APIC_BASE MSR (see Figure 12-5) of the BSP, indicating that it is the BSP. This flag is cleared for all other processors.

The BSP executes the BIOS's boot-strap code to configure the APIC environment, sets up system-wide data structures, and starts and initializes the APs. When the BSP and APs are initialized, the BSP then begins executing the operating-system initialization code.

Following a power-up or reset, the APs complete a minimal self-configuration, then wait for a startup signal (a SIPI message) from the BSP processor. Upon receiving a SIPI message, an AP executes the BIOS AP configuration code, which ends with the AP being placed in halt state.

For Intel 64 and IA-32 processors supporting Intel Hyper-Threading Technology, the MP initialization protocol treats each of the logical processors on the system bus or coherent link domain as a separate processor (with a unique

APIC ID). During boot-up, one of the logical processors is selected as the BSP and the remainder of the logical processors are designated as APs.

10.4.2 MP Initialization Protocol Requirements and Restrictions

The MP initialization protocol imposes the following requirements and restrictions on the system:

- The MP protocol is executed only after a power-up or RESET. If the MP protocol has completed and a BSP is chosen, subsequent INITs (either to a specific processor or system wide) do not cause the MP protocol to be repeated. Instead, each logical processor examines its BSP flag (in the IA32_APIC_BASE MSR) to determine whether it should execute the BIOS boot-strap code (if it is the BSP) or enter a wait-for-SIPI state (if it is an AP).
- All devices in the system that are capable of delivering interrupts to the processors must be inhibited from doing so for the duration of the MP initialization protocol. The time during which interrupts must be inhibited includes the window between when the BSP issues an INIT-SIPI-SIPI sequence to an AP and when the AP responds to the last SIPI in the sequence.

10.4.3 MP Initialization Protocol Algorithm for MP Systems

Following a power-up or RESET of an MP system, the processors in the system execute the MP initialization protocol algorithm to initialize each of the logical processors on the system bus or coherent link domain. In the course of executing this algorithm, the following boot-up and initialization operations are carried out:

1. Each logical processor is assigned a unique APIC ID, based on system topology. The unique ID is a 32-bit value if the processor supports CPUID leaf 0BH, otherwise the unique ID is an 8-bit value. (see Section 10.4.5, "Identifying Logical Processors in an MP System").
2. Each logical processor is assigned a unique arbitration priority based on its APIC ID.
3. Each logical processor executes its internal BIST simultaneously with the other logical processors in the system.
4. Upon completion of the BIST, the logical processors use a hardware-defined selection mechanism to select the BSP and the APs from the available logical processors on the system bus. The BSP selection mechanism differs depending on the family, model, and stepping IDs of the processors, as follows:
   - Later generations of IA processors within family 0FH (see Section 10.4), IA processors with system bus (family=06H, extended_model=0, model>=0EH), or all other modern Intel processors (family=06H, extended_model>0):
     - The logical processors begin monitoring the BNR# signal, which is toggling. When the BNR# pin stops toggling, each processor attempts to issue a NOP special cycle on the system bus.
     - The logical processor with the highest arbitration priority succeeds in issuing a NOP special cycle and is nominated the BSP. This processor sets the BSP flag in its IA32_APIC_BASE MSR, then fetches and begins executing BIOS boot-strap code, beginning at the reset vector (physical address FFFF FFF0H).
     - The remaining logical processors (that failed in issuing a NOP special cycle) are designated as APs. They leave their BSP flags in the clear state and enter a "wait-for-SIPI state."
   - Early generations of IA processors within family 0FH (family=0FH, model=0H, stepping<=09H), P6 family or older processors supporting MP operations (family=06H, extended_model=0, model<=0DH; or family <06H):
     - Each processor broadcasts a BIPI to "all including self." The first processor that broadcasts a BIPI (and thus receives its own BIPI vector), selects itself as the BSP and sets the BSP flag in its IA32_APIC_BASE MSR. (See Section 10.11.1, "Overview of the MP Initialization Process for P6 Family Processors," for a description of the BIPI, FIPI, and SIPI messages.)
     - The remainder of the processors (which were not selected as the BSP) are designated as APs. They leave their BSP flags in the clear state and enter a "wait-for-SIPI state."
     - The newly established BSP broadcasts an FIPI message to "all including self," which the BSP and APs treat as an end of MP initialization signal. Only the processor with its BSP flag set responds to the FIPI

message. It responds by fetching and executing the BIOS boot-strap code, beginning at the reset vector (physical address FFFF FFF0H).

5. As part of the boot-strap code, the BSP creates an ACPI table and/or an MP table and adds its initial APIC ID to these tables as appropriate.
6. At the end of the boot-strap procedure, the BSP sets a processor counter to 1, then broadcasts a SIPI message to all the APs in the system. Here, the SIPI message contains a vector to the BIOS AP initialization code (at 000VV000H, where VV is the vector contained in the SIPI message).
7. The first action of the AP initialization code is to set up a race (among the APs) to a BIOS initialization semaphore. The first AP to the semaphore begins executing the initialization code. (See Section 10.4.4, "MP Initialization Example," for semaphore implementation details.) As part of the AP initialization procedure, the AP adds its APIC ID number to the ACPI and/or MP tables as appropriate and increments the processor counter by 1. At the completion of the initialization procedure, the AP executes a CLI instruction and halts itself.
8. When each of the APs has gained access to the semaphore and executed the AP initialization code, the BSP establishes a count for the number of processors connected to the system bus, completes executing the BIOS boot-strap code, and then begins executing operating-system boot-strap and start-up code.
9. While the BSP is executing operating-system boot-strap and start-up code, the APs remain in the halted state. In this state they will respond only to INITs, NMIs, and SMIs. They will also respond to snoops and to assertions of the STPCLK# pin.

The following section gives an example (with code) of the MP initialization protocol for of multiple processors operating in an MP configuration.

Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, describes how to program the LINT[0:1] pins of the processor's local APICs after an MP configuration has been completed.

10.4.4 MP Initialization Example

The following example illustrates the use of the MP initialization protocol used to initialize processors in an MP system after the BSP and APs have been established. The code runs on Intel 64 or IA-32 processors that use a protocol. This includes P6 Family processors, Pentium 4 processors, Intel Core Duo, Intel Core 2 Duo and Intel Xeon processors.

The following constants and data definitions are used in the accompanying
code examples. They are based on the addresses of the APIC registers defined in Table 12-1.

```
ICR_LOW         EQU 0FEE00300H
SVR             EQU 0FEE000F0H
APIC_ID         EQU 0FEE00020H
LVT3            EQU 0FEE00370H
APIC_ENABLED    EQU 0100H
BOOT_ID         DD ?
COUNT           EQU 00H
VACANT          EQU 00H
```

10.4.4.1 Typical BSP Initialization Sequence

After the BSP and APs have been selected (by means of a hardware protocol, see Section 10.4.3, "MP Initialization Protocol Algorithm for MP Systems"), the BSP begins executing BIOS boot-strap code (POST) at the normal IA-32 architecture starting address (FFFF FFF0H). The boot-strap code typically performs the following operations:

1. Initializes memory.
2. Loads the microcode update into the processor.
3. Initializes the MTRRs.
4. Enables the caches.

5. Executes the CPUID instruction with a value of 0H in the EAX register, then reads the EBX, ECX, and EDX registers to determine if the BSP is "GenuineIntel."
6. Executes the CPUID instruction with a value of 1H in the EAX register, then saves the values in the EAX, ECX, and EDX registers in a system configuration space in RAM for use later.
7. Loads start-up code for the AP to execute into a 4-KByte page in the lower 1 MByte of memory.
8. Switches to protected mode and ensures that the APIC address space is mapped to the strong uncacheable (UC) memory type.
9. Determine the BSP's APIC ID from the local APIC ID register (default is 0), the code snippet below is an example that applies to logical processors in a system whose local APIC units operate in xAPIC mode that APIC registers are accessed using memory mapped interface:

```
MOV ESI, APIC_ID; Address of local APIC ID register
MOV EAX, [ESI];
AND EAX, 0FF000000H; Zero out all other bits except APIC ID
MOV BOOT_ID, EAX; Save in memory
```

   Saves the APIC ID in the ACPI and/or MP tables and optionally in the system configuration space in RAM.
10. Converts the base address of the 4-KByte page for the AP's bootup code into 8-bit vector. The 8-bit vector defines the address of a 4-KByte page in the real-address mode address space (1-MByte space). For example, a vector of 0BDH specifies a start-up memory address of 000BD000H.
11. Enables the local APIC by setting bit 8 of the APIC spurious vector register (SVR).

```
MOV ESI, SVR; Address of SVR
MOV EAX, [ESI];
OR  EAX, APIC_ENABLED; Set bit 8 to enable (0 on reset)
MOV [ESI], EAX;
```

12. Sets up the LVT error handling entry by establishing an 8-bit vector for the APIC error handler.

```
MOV ESI, LVT3;
MOV EAX, [ESI];
AND EAX, FFFFFF00H; Clear out previous vector.
OR EAX, 000000xxH; xx is the 8-bit vector the APIC error handler.
MOV [ESI], EAX;
```

13. Initializes the Lock Semaphore variable VACANT to 00H. The APs use this semaphore to determine the order in which they execute BIOS AP initialization code.
14. Performs the following operation to set up the BSP to detect the presence of APs in the system and the number of processors (within a finite duration, minimally 100 milliseconds):
    — Sets the value of the COUNT variable to 1.
    — In the AP BIOS initialization code, the AP will increment the COUNT variable to indicate its presence. The finite duration while waiting for the COUNT to be updated can be accomplished with a timer. When the timer expires, the BSP checks the value of the COUNT variable. If the timer expires and the COUNT variable has not been incremented, no APs are present or some error has occurred.
15. Broadcasts an INIT-SIPI-SIPI IPI sequence to the APs to wake them up and initialize them. Alternatively, following a power-up or RESET, since all APs are already in the "wait-for-SIPI state," the BSP can broadcast just a single SIPI IPI to the APs to wake them up and initialize them. If software knows how many logical processors it expects to wake up, it may choose to poll the COUNT variable. If the expected processors show up before the 100 millisecond timer expires, the timer can be canceled and skip to step 16.

    The left-hand-side of the procedure illustrated in Table 10-1 provides an algorithm when the expected processor count is unknown. The right-hand-side of Table 10-1 can be used when the expected processor count is known.

Table 10-1. Broadcast INIT-SIPI-SIPI Sequence and Choice of Timeouts

INIT-SIPI-SIPI when the expected processor count is unknown	INIT-SIPI-SIPI when the expected processor count is known

MOV ESI, ICR_LOW; Load address of ICR low dword into ESI.
MOV EAX, 000C4500H; Load ICR encoding for broadcast INIT IPI
; to all APs into EAX.
MOV [ESI], EAX; Broadcast INIT IPI to all APs
; 10-millisecond delay loop.
MOV EAX, 000C46XXH; Load ICR encoding for broadcast SIPI IP
; to all APs into EAX, where xx is the vector computed in step 10.
MOV [ESI], EAX; Broadcast SIPI IPI to all APs
; 200-microsecond delay loop
MOV [ESI], EAX; Broadcast second SIPI IPI to all APs
;Waits for the timer interrupt until the timer expires
	MOV ESI, ICR_LOW; Load address of ICR low dword into ESI.
	MOV EAX, 000C4500H; Load ICR encoding for broadcast INIT IPI
	; to all APs into EAX.
	MOV [ESI], EAX; Broadcast INIT IPI to all APs
	; 10-millisecond delay loop.
	MOV EAX, 000C46XXH; Load ICR encoding for broadcast SIPI IP
	; to all APs into EAX, where xx is the vector computed in step 10.
	MOV [ESI], EAX; Broadcast SIPI IPI to all APs
	; 200 microsecond delay loop with check to see if COUNT has
	; reached the expected processor count. If COUNT reaches
	; expected processor count, cancel timer and go to step 16.
	MOV [ESI], EAX; Broadcast second SIPI IPI to all APs
	; Wait for the timer interrupt polling COUNT. If COUNT reaches
	; expected processor count, cancel timer and go to step 16.
	; If timer expires, go to step 16.

16. Reads and evaluates the COUNT variable and establishes a processor count.
17. If necessary, reconfigures the APIC and continues with the remaining system diagnostics as appropriate.

10.4.4.2 Typical AP Initialization Sequence

When an AP receives the SIPI, it begins executing BIOS AP initialization code at the vector encoded in the SIPI. The AP initialization code typically performs the following operations:

1. Waits on the BIOS initialization Lock Semaphore. When control of the semaphore is attained, initialization continues.
2. Loads the microcode update into the processor.
3. Initializes the MTRRs (using the same mapping that was used for the BSP).
4. Enables the cache.
5. Executes the CPUID instruction with a value of 0H in the EAX register, then reads the EBX, ECX, and EDX registers to determine if the AP is "GenuineIntel."
6. Executes the CPUID instruction with a value of 1H in the EAX register, then saves the values in the EAX, ECX, and EDX registers in a system configuration space in RAM for use later.
7. Switches to protected mode and ensures that the APIC address space is mapped to the strong uncacheable (UC) memory type.
8. Determines the AP's APIC ID from the local APIC ID register, and adds it to the MP and ACPI tables and optionally to the system configuration space in RAM.
9. Initializes and configures the local APIC by setting bit 8 in the SVR register and setting up the LVT3 (error LVT) for error handling (as described in steps 9 and 10 in Section 10.4.4.1, "Typical BSP Initialization Sequence").
10. Configures the APs SMI execution environment. (Each AP and the BSP must have a different SMBASE address.)
11. Increments the COUNT variable by 1.
12. Releases the semaphore.
13. Executes one of the following:

— the CLI and HLT instructions (if MONITOR/MWAIT is not supported), or

— the CLI, MONITOR, and MWAIT sequence to enter a deep C-state.

14. Waits for an INIT IPI.

10.4.5 Identifying Logical Processors in an MP System

After the BIOS has completed the MP initialization protocol, each logical processor can be uniquely identified by its local APIC ID. Software can access these APIC IDs in either of the following ways:

- **Read APIC ID for a local APIC** — Code running on a logical processor can read APIC ID in one of two ways depending on the local APIC unit is operating in x2APIC mode or in xAPIC mode:
  - If the local APIC unit supports x2APIC and is operating in x2APIC mode, 32-bit APIC ID can be read by executing a RDMSR instruction to read the processor's x2APIC ID register. This method is equivalent to executing CPUID leaf 0BH described below.
  - If the local APIC unit is operating in xAPIC mode, 8-bit APIC ID can be read by executing a MOV instruction to read the processor's local APIC ID register (see Section 12.4.6, "Local APIC ID"). This is the ID to use for directing physical destination mode interrupts to the processor.
- **Read ACPI or MP table** — As part of the MP initialization protocol, the BIOS creates an ACPI table and an MP table. These tables are defined in the Multiprocessor Specification Version 1.4 and provide software with a list of the processors in the system and their local APIC IDs. The format of the ACPI table is derived from the ACPI specification, which is an industry standard power management and platform configuration specification for MP systems.
- **Read Initial APIC ID** (If the processor does not support CPUID leaf 0BH) — An APIC ID is assigned to a logical processor during power up. This is the initial APIC ID reported by CPUID.1:EBX[31:24] and may be different from the current value read from the local APIC. The initial APIC ID can be used to determine the topological relationship between logical processors for multi-processor systems that do not support CPUID leaf 0BH.

  Bits in the 8-bit initial APIC ID can be interpreted using several bit masks. Each bit mask can be used to extract an identifier to represent a hierarchical domain of the multi-threading resource topology in an MP system (See Section 10.9.1, "Hierarchical Mapping of Shared Resources"). The initial APIC ID may consist of up to four bit-fields. In a non-clustered MP system, the field consists of up to three bit fields.
- **Read 32-bit APIC ID from CPUID leaf 0BH** (If the processor supports CPUID leaf 0BH) — A unique APIC ID is assigned to a logical processor during power up. This APIC ID is reported by CPUID.0BH:EDX[31:0] as a 32-bit value. Use the 32-bit APIC ID and CPUID leaf 0BH to determine the topological relationship between logical processors if the processor supports CPUID leaf 0BH.

  Bits in the 32-bit x2APIC ID can be extracted into sub-fields using CPUID leaf 0BH parameters. (See Section 10.9.1, "Hierarchical Mapping of Shared Resources").

Figure 10-2 shows two examples of APIC ID bit fields in earlier single-core processors. In single-core Intel Xeon processors, the APIC ID assigned to a logical processor during power-up and initialization is 8 bits. Bits 2:1 form a 2-bit physical package identifier (which can also be thought of as a socket identifier). In systems that configure physical processors in clusters, bits 4:3 form a 2-bit cluster ID. Bit 0 is used in the Intel Xeon processor MP to identify the two logical processors within the package (see Section 10.9.3, "Hierarchical ID of Logical Processors in an MP System"). For Intel Xeon processors that do not support Intel Hyper-Threading Technology, bit 0 is always set to 0; for Intel Xeon processors supporting Intel Hyper-Threading Technology, bit 0 performs the same function as it does for Intel Xeon processor MP.

For more recent multi-core processors, see Section 10.9.1, "Hierarchical Mapping of Shared Resources," for a complete description of the topological relationships between logical processors and bit field locations within an initial APIC ID across Intel 64 and IA-32 processor families.

Note the number of bit fields and the width of bit-fields are dependent on processor and platform hardware capabilities. Software should determine these at runtime. When initial APIC IDs are assigned to logical processors, the value of APIC ID assigned to a logical processor will respect the bit-field boundaries corresponding core, physical package, etc. Additional examples of the bit fields in the initial APIC ID of multi-threading capable systems are shown in Section 10.9.

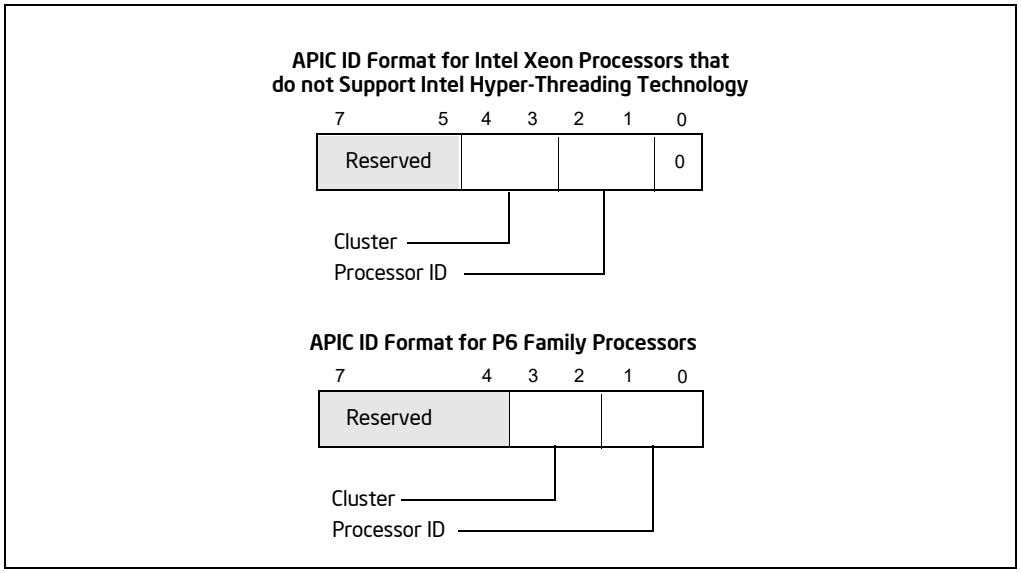


Figure 10-2. Interpretation of APIC ID in Early MP Systems

For P6 family processors, the APIC ID that is assigned to a processor during power-up and initialization is 4 bits (see Figure 10-2). Here, bits 0 and 1 form a 2-bit processor (or socket) identifier and bits 2 and 3 form a 2-bit cluster ID.

10.5 INTEL® HYPER-THREADING TECHNOLOGY AND INTEL® MULTI-CORE TECHNOLOGY

Intel Hyper-Threading Technology and Intel multi-core technology are extensions to Intel 64 and IA-32 architectures that enable a single physical processor to execute two or more separate code streams (called *threads*) concurrently. In Intel Hyper-Threading Technology, a single processor core provides two logical processors that share execution resources (see Section 10.7, "Intel® Hyper-Threading Technology Architecture"). In Intel multi-core technology, a physical processor package provides two or more processor cores. Both configurations require chipsets and a BIOS that support the technologies.

Software should not rely on processor names to determine whether a processor supports Intel Hyper-Threading Technology or Intel multi-core technology. Use the CPUID instruction to determine processor capability (see Section 10.6.2, "Initializing Multi-Core Processors").

10.6 DETECTING HARDWARE MULTI-THREADING SUPPORT AND TOPOLOGY

Use the CPUID instruction to detect the presence of hardware multi-threading support in a physical processor. Hardware multi-threading can support several varieties of multigrade and/or Intel Hyper-Threading Technology. CPUID instruction provides several sets of parameter information to aid software enumerating topology information. The relevant topology enumeration parameters provided by CPUID include:

- **Hardware Multi-Threading feature flag (CPUID.1:EDX[28] = 1)** — Indicates when set that the physical package is capable of supporting Intel Hyper-Threading Technology and/or multiple cores.
- **Processor topology enumeration parameters for 8-bit APIC ID**:
  - **Addressable IDs for Logical processors in the same Package (CPUID.1:EBX[23:16])** — Indicates the maximum number of addressable ID for logical processors in a physical package. Within a physical package, there may be addressable IDs that are not occupied by any logical processors. This parameter does not represents the hardware capability of the physical processor.[1]

- **Addressable IDs for processor cores in the same Package[1] (CPUID.(EAX=4, ECX=0[2]):EAX[31:26] + 1 = Y)** — Indicates the maximum number of addressable IDs attributable to processor cores (Y) in the physical package.
- **Extended Processor Topology Enumeration parameters for 32-bit APIC ID**: Intel 64 processors supporting CPUID leaf 0BH will assign unique APIC IDs to each logical processor in the system. CPUID leaf 0BH reports the 32-bit APIC ID and provide topology enumeration parameters. See CPUID instruction reference pages in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

The CPUID feature flag may indicate support for hardware multi-threading when only one logical processor available in the package. In this case, the decimal value represented by bits 16 through 23 in the EBX register will have a value of 1.

Software should note that the number of logical processors enabled by system software may be less than the value of "Addressable IDs for Logical processors". Similarly, the number of cores enabled by system software may be less than the value of "Addressable IDs for processor cores".

Software can detect the availability of the CPUID extended topology enumeration leaf (0BH) by performing two steps:

- Check maximum input value for basic CPUID information by executing CPUID with EAX= 0. If CPUID.0H:EAX is greater than or equal or 11 (0BH), then proceed to next step,
- Check CPUID.EAX=0BH, ECX=0H:EBX is non-zero.

If both of the above conditions are true, extended topology enumeration leaf is available. Note the presence of CPUID leaf 0BH in a processor does not guarantee support that the local APIC supports x2APIC. If CPUID.(EAX=0BH, ECX=0H):EBX returns zero and maximum input value for basic CPUID information is greater than 0BH, then CPUID.0BH leaf is not supported on that processor.

10.6.1 Initializing Processors Supporting Intel® Hyper-Threading Technology

The initialization process for an MP system that contains processors supporting Intel Hyper-Threading Technology is the same as for conventional MP systems (see Section 10.4, "Multiple-Processor (MP) Initialization"). One logical processor in the system is selected as the BSP and other processors (or logical processors) are designated as APs. The initialization process is identical to that described in Section 10.4.3, "MP Initialization Protocol Algorithm for MP Systems," and Section 10.4.4, "MP Initialization Example."

During initialization, each logical processor is assigned an APIC ID that is stored in the local APIC ID register for each logical processor. If two or more processors supporting Intel Hyper-Threading Technology are present, each logical processor on the system bus is assigned a unique ID (see Section 10.9.3, "Hierarchical ID of Logical Processors in an MP System"). Once logical processors have APIC IDs, software communicates with them by sending APIC IPI messages.

10.6.2 Initializing Multi-Core Processors

The initialization process for an MP system that contains multi-core Intel 64 or IA-32 processors is the same as for conventional MP systems (see Section 10.4, "Multiple-Processor (MP) Initialization"). A logical processor in one core is selected as the BSP; other logical processors are designated as APs.

During initialization, each logical processor is assigned an APIC ID. Once logical processors have APIC IDs, software may communicate with them by sending APIC IPI messages.

---

1. Operating system and BIOS may implement features that reduce the number of logical processors available in a platform to applications at runtime to less than the number of physical packages times the number of hardware-capable logical processors per package.

1. Software must check CPUID for its support of leaf 4 when implementing support for multi-core. If CPUID leaf 4 is not available at runtime, software should handle the situation as if there is only one core per package.

2. Maximum number of cores in the physical package must be queried by executing CPUID with EAX=4 and a valid ECX input value. Valid ECX input values start from 0.

10.6.3 Executing Multiple Threads on an Intel® 64 or IA-32 Processor Supporting Hardware Multi-Threading

Upon completing the operating system boot-up procedure, the bootstrap processor (BSP) executes operating system code. Other logical processors are placed in the halt state. To execute a code stream (thread) on a halted logical processor, the operating system issues an interprocessor interrupt (IPI) addressed to the halted logical processor. In response to the IPI, the processor wakes up and begins executing the code identified by the vector received as part of the IPI.

To manage execution of multiple threads on logical processors, an operating system can use conventional symmetric multiprocessing (SMP) techniques. For example, the operating-system can use a time-slice or load balancing mechanism to periodically interrupt each of the active logical processors. Upon interrupting a logical processor, the operating system checks its run queue for a thread waiting to be executed and dispatches the thread to the interrupted logical processor.

10.6.4 Handling Interrupts on an IA-32 Processor Supporting Hardware Multi-Threading

Interrupts are handled on processors supporting Intel Hyper-Threading Technology as they are on conventional MP systems. External interrupts are received by the I/O APIC, which distributes them as interrupt messages to specific logical processors (see Figure 10-3).

Logical processors can also send IPIs to other logical processors by writing to the ICR register of its local APIC (see Section 12.6, “Issuing Interprocessor Interrupts”). This also applies to dual-core processors.

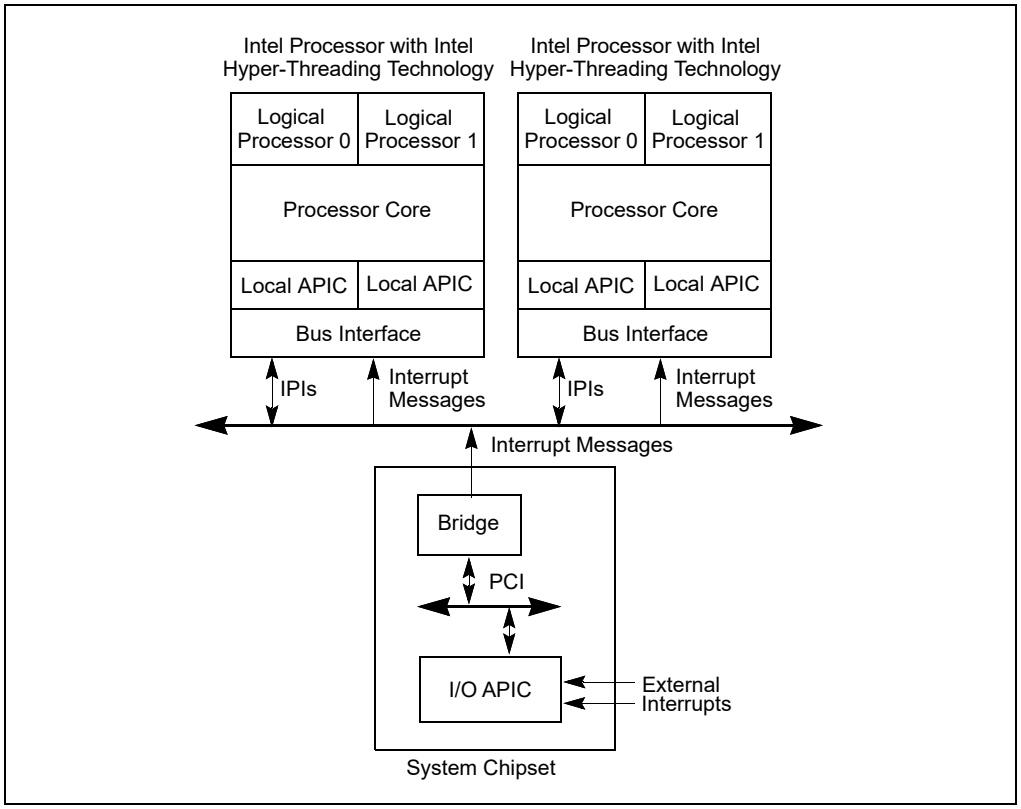


Figure 10-3. Local APICs and I/O APIC in MP System Supporting Intel HT Technology

10.7 INTEL® HYPER-THREADING TECHNOLOGY ARCHITECTURE

Figure 10-4 shows a generalized view of an Intel processor supporting Intel Hyper-Threading Technology, using the original Intel Xeon processor MP as an example. This implementation of the Intel Hyper-Threading Technology

consists of two logical processors (each represented by a separate architectural state) which share the processor's execution engine and the bus interface. Each logical processor also has its own advanced programmable interrupt controller (APIC).

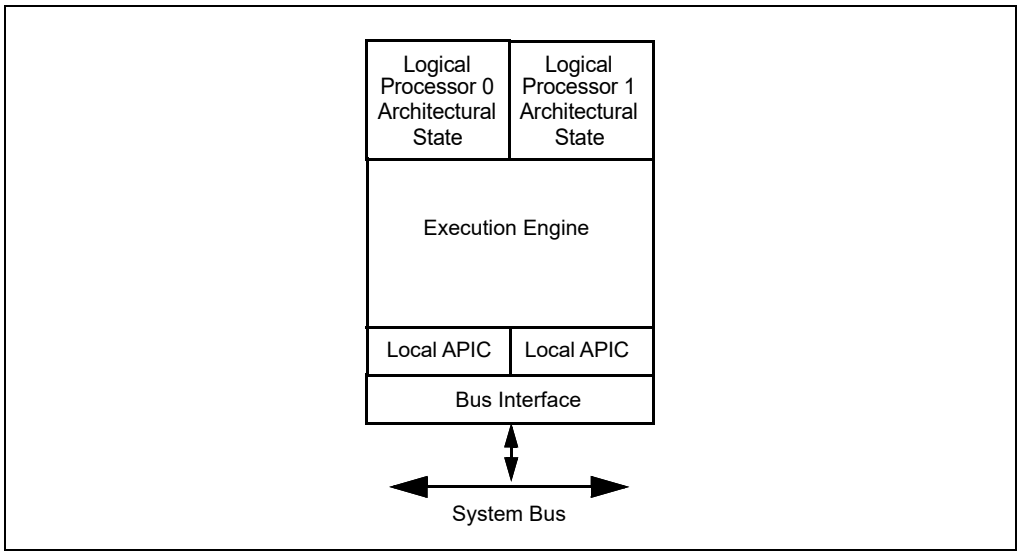


Figure 10-4. IA-32 Processor with Two Logical Processors Supporting Intel HT Technology

10.7.1 State of the Logical Processors

The following features are part of the architectural state of logical processors within Intel 64 or IA-32 processors supporting Intel Hyper-Threading Technology. The features can be subdivided into three groups:

- Duplicated for each logical processor
- Shared by logical processors in a physical processor
- Shared or duplicated, depending on the implementation

The following features are duplicated for each logical processor:

- General purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP)
- Segment registers (CS, DS, SS, ES, FS, and GS)
- EFLAGS and EIP registers. Note that the CS and EIP/RIP registers for each logical processor point to the instruction stream for the thread being executed by the logical processor.
- x87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM7) and the MXCSR register
- Control registers and system table pointer registers (GDTR, LDTR, IDTR, task register)
- Debug registers (DR0, DR1, DR2, DR3, DR6, DR7) and the debug control MSRs
- Machine check global status (IA32_MCG_STATUS) and machine check capability (IA32_MCG_CAP) MSRs
- Thermal clock modulation and ACPI Power management control MSRs
- Time stamp counter MSRs
- Most of the other MSR registers, including the page attribute table (PAT). See the exceptions below.
- Local APIC registers.
- Additional general purpose registers (R8-R15), XMM registers (XMM8-XMM15), control register, IA32_EFER on Intel 64 processors.

The following features are shared by logical processors:

- Memory type range registers (MTRRs)

Whether the following features are shared or duplicated is implementation-specific:

- IA32_MISC_ENABLE MSR (MSR address 1A0H)
- Machine check architecture (MCA) MSRs (except for the IA32_MCG_STATUS and IA32_MCG_CAP MSRs)
- Performance monitoring control and counter MSRs

10.7.2 APIC Functionality

When a processor supporting Intel Hyper-Threading Technology support is initialized, each logical processor is assigned a local APIC ID (see Table 12-1). The local APIC ID serves as an ID for the logical processor and is stored in the logical processor's APIC ID register. If two or more processors supporting Intel Hyper-Threading Technology are present in a dual processor (DP) or MP system, each logical processor on the system bus is assigned a unique local APIC ID (see Section 10.9.3, "Hierarchical ID of Logical Processors in an MP System").

Software communicates with local processors using the APIC's interprocessor interrupt (IPI) messaging facility. Setup and programming for APICs is identical in processors that support and do not support Intel Hyper-Threading Technology. See Chapter 12, "Advanced Programmable Interrupt Controller (APIC)," for a detailed discussion.

10.7.3 Memory Type Range Registers (MTRR)

MTRRs in a processor supporting Intel Hyper-Threading Technology are shared by logical processors. When one logical processor updates the setting of the MTRRs, settings are automatically shared with the other logical processors in the same physical package.

The architectures require that all MP systems based on Intel 64 and IA-32 processors (this includes logical processors) must use an identical MTRR memory map. This gives software a consistent view of memory, independent of the processor on which it is running. See Section 13.11, "Memory Type Range Registers (MTRRs)," for information on setting up MTRRs.

10.7.4 Page Attribute Table (PAT)

Each logical processor has its own PAT MSR (IA32_PAT). However, as described in Section 13.12, "Page Attribute Table (PAT)," the PAT MSR settings must be the same for all processors in a system, including the logical processors.

10.7.5 Machine Check Architecture

In the Intel HT Technology context as implemented by processors based on Intel NetBurst® microarchitecture, all of the machine check architecture (MCA) MSRs (except for the IA32_MCG_STATUS and IA32_MCG_CAP MSRs) are duplicated for each logical processor. This permits logical processors to initialize, configure, query, and handle machine-check exceptions simultaneously within the same physical processor. The design is compatible with machine check exception handlers that follow the guidelines given in Chapter 17, "Machine-Check Architecture."

The IA32_MCG_STATUS MSR is duplicated for each logical processor so that its machine check in progress bit field (MCIP) can be used to detect recursion on the part of MCA handlers. In addition, the MSR allows each logical processor to determine that a machine-check exception is in progress independent of the actions of another logical processor in the same physical package.

Because the logical processors within a physical package are tightly coupled with respect to shared hardware resources, both logical processors are notified of machine check errors that occur within a given physical processor. If machine-check exceptions are enabled when a fatal error is reported, all the logical processors within a physical package are dispatched to the machine-check exception handler. If machine-check exceptions are disabled, the logical processors enter the shutdown state and assert the IERR# signal.

When enabling machine-check exceptions, the MCE flag in control register CR4 should be set for each logical processor.

On Intel Atom family processors that support Intel Hyper-Threading Technology, the MCA facilities are shared between all logical processors on the same processor core.

10.7.6 Debug Registers and Extensions

Each logical processor has its own set of debug registers (DR0, DR1, DR2, DR3, DR6, DR7) and its own debug control MSR. These can be set to control and record debug information for each logical processor independently. Each logical processor also has its own last branch records (LBR) stack.

10.7.7 Performance Monitoring Counters

Performance counters and their companion control MSRs are shared between the logical processors within a processor core for processors based on Intel NetBurst microarchitecture. As a result, software must manage the use of these resources. The performance counter interrupts, events, and precise event monitoring support can be set up and allocated on a per thread (per logical processor) basis.

See Section 21.6.4, "Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture," for a discussion of performance monitoring in the Intel Xeon processor MP.

In Intel Atom processor family that support Intel Hyper-Threading Technology, the performance counters (general-purpose and fixed-function counters) and their companion control MSRs are duplicated for each logical processor.

10.7.8 IA32_MISC_ENABLE MSR

The IA32_MISC_ENABLE MSR (MSR address 1A0H) is generally shared between the logical processors in a processor core supporting Intel Hyper-Threading Technology. However, some bit fields within IA32_MISC_ENABLE MSR may be duplicated per logical processor. The partition of shared or duplicated bit fields within IA32_MISC_ENABLE is implementation dependent. Software should program duplicated fields carefully on all logical processors in the system to ensure consistent behavior.

10.7.9 Memory Ordering

The logical processors in an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology obey the same rules for memory ordering as Intel 64 or IA-32 processors without Intel HT Technology (see Section 10.2, "Memory Ordering"). Each logical processor uses a processor-ordered memory model that can be further defined as "write-ordered with store buffer forwarding." All mechanisms for strengthening or weakening the memory-ordering model to handle special programming situations apply to each logical processor.

10.7.10 Serializing Instructions

As a general rule, when a logical processor in a processor supporting Intel Hyper-Threading Technology executes a serializing instruction, only that logical processor is affected by the operation. An exception to this rule is the execution of the WBINVD, INVD, and WRMSR instructions; and the MOV CR instruction when the state of the CD flag in control register CR0 is modified. Here, both logical processors are serialized.

10.7.11 Microcode Update Resources

In an Intel processor supporting Intel Hyper-Threading Technology, the microcode update facilities are shared between the logical processors; either logical processor can initiate an update. Each logical processor has its own BIOS signature MSR (IA32_BIOS_SIGN_ID at MSR address 8BH). When a logical processor performs an update for the physical processor, the IA32_BIOS_SIGN_ID MSRs for resident logical processors are updated with identical information. If logical processors initiate an update simultaneously, the processor core provides the necessary synchronization needed to ensure that only one update is performed at a time.

NOTE

Some processors (prior to the introduction of Intel 64 Architecture and based on Intel NetBurst microarchitecture) do not support simultaneous loading of microcode update to the sibling logical processors in the same core. All other processors support logical processors initiating an update simultaneously. Intel recommends a common approach that the microcode loader use the sequential technique described in Section 11.11.6.3.

10.7.12 Self Modifying Code

Intel processors supporting Intel Hyper-Threading Technology support self-modifying code, where data writes modify instructions cached or currently in flight. They also support cross-modifying code, where on an MP system writes generated by one processor modify instructions cached or currently in flight on another. See Section 10.1.3, "Handling Self- and Cross-Modifying Code," for a description of the requirements for self- and cross-modifying code in an IA-32 processor.

10.7.13 Implementation-Specific Intel® HT Technology Facilities

The following non-architectural facilities are implementation-specific in IA-32 processors supporting Intel Hyper-Threading Technology:

- Caches.
- Translation lookaside buffers (TLBs).
- Thermal monitoring facilities.

The Intel Xeon processor MP implementation is described in the following sections.

10.7.13.1 Processor Caches

For processors supporting Intel Hyper-Threading Technology, the caches are shared. Any cache manipulation instruction that is executed on one logical processor has a global effect on the cache hierarchy of the physical processor. Note the following:

- **WBINVD instruction** — The entire cache hierarchy is invalidated after modified data is written back to memory. All logical processors are stopped from executing until after the write-back and invalidate operation is completed. A special bus cycle is sent to all caching agents. The amount of time or cycles for WBINVD to complete will vary due to the size of different cache hierarchies and other factors. As a consequence, the use of the WBINVD instruction can have an impact on interrupt/event response time.
- **INVD instruction** — The entire cache hierarchy is invalidated without writing back modified data to memory. All logical processors are stopped from executing until after the invalidate operation is completed. A special bus cycle is sent to all caching agents.
- **CLFLUSH and CLFLUSHOPT instructions** — The specified cache line is invalidated from the cache hierarchy after any modified data is written back to memory and a bus cycle is sent to all caching agents, regardless of which logical processor caused the cache line to be filled.
- **CD flag in control register CR0** — Each logical processor has its own CR0 control register, and thus its own CD flag in CR0. The CD flags for the two logical processors are ORed together, such that when any logical processor sets its CD flag, the entire cache is nominally disabled.

10.7.13.2 Processor Translation Lookaside Buffers (TLBs)

In processors supporting Intel Hyper-Threading Technology, data cache TLBs are shared. The instruction cache TLB may be duplicated or shared in each logical processor, depending on implementation specifics of different processor families.

Entries in the TLBs are tagged with an ID that indicates the logical processor that initiated the translation. This tag applies even for translations that are marked global using the page-global feature for memory paging. See Section 5.10, "Caching Translation Information," for information about global translations.

When a logical processor performs a TLB invalidation operation, only the TLB entries that are tagged for that logical processor are guaranteed to be flushed. This protocol applies to all TLB invalidation operations, including writes to control registers CR3 and CR4 and uses of the INVLPG instruction.

10.7.13.3 Thermal Monitor

In a processor that supports Intel Hyper-Threading Technology, logical processors share the catastrophic shutdown detector and the automatic thermal monitoring mechanism (see Section 16.8, "Thermal Monitoring and Protection"). Sharing results in the following behavior:

- If the processor's core temperature rises above the preset catastrophic shutdown temperature, the processor core halts execution, which causes both logical processors to stop execution.
- When the processor's core temperature rises above the preset automatic thermal monitor trip temperature, the frequency of the processor core is automatically modulated, which effects the execution speed of both logical processors.

For software controlled clock modulation, each logical processor has its own IA32_CLOCK_MODULATION MSR, allowing clock modulation to be enabled or disabled on a logical processor basis. Typically, if software controlled clock modulation is going to be used, the feature must be enabled for all the logical processors within a physical processor and the modulation duty cycle must be set to the same value for each logical processor. If the duty cycle values differ between the logical processors, the processor clock will be modulated at the highest duty cycle selected.

10.7.13.4 External Signal Compatibility

This section describes the constraints on external signals received through the pins of a processor supporting Intel Hyper-Threading Technology and how these signals are shared between its logical processors.

- **STPCLK#** — A single STPCLK# pin is provided on the physical package of the Intel Xeon processor MP. External control logic uses this pin for power management within the system. When the STPCLK# signal is asserted, the processor core transitions to the stop-grant state, where instruction execution is halted but the processor core continues to respond to snoop transactions. Regardless of whether the logical processors are active or halted when the STPCLK# signal is asserted, execution is stopped on both logical processors and neither will respond to interrupts.

  In MP systems, the STPCLK# pins on all physical processors are generally tied together. As a result this signal affects all the logical processors within the system simultaneously.
- **LINT0 and LINT1 pins** — A processor supporting Intel Hyper-Threading Technology has only one set of LINT0 and LINT1 pins, which are shared between the logical processors. When one of these pins is asserted, both logical processors respond unless the pin has been masked in the APIC local vector tables for one or both of the logical processors.

  Typically in MP systems, the LINT0 and LINT1 pins are not used to deliver interrupts to the logical processors. Instead all interrupts are delivered to the local processors through the I/O APIC.
- **A20M# pin** — On an IA-32 processor, the A20M# pin is typically provided for compatibility with the Intel 286 processor. Asserting this pin causes bit 20 of the physical address to be masked (forced to zero) for all external bus memory accesses. Processors supporting Intel Hyper-Threading Technology provide one A20M# pin, which affects the operation of both logical processors within the physical processor.

  The functionality of A20M# is used primarily by older operating systems and not used by modern operating systems. On newer Intel 64 processors, A20M# may be absent.

10.8 MULTI-CORE ARCHITECTURE

This section describes the architecture of Intel 64 and IA-32 processors supporting dual-core and quad-core technology. The discussion is applicable to the Intel Pentium processor Extreme Edition, Pentium D, Intel Core Duo, Intel Core 2 Duo, Dual-core Intel Xeon processor, Intel Core 2 Quad processors, and quad-core Intel Xeon processors. Features vary across different microarchitectures and are detectable using CPUID.

In general, each processor core has dedicated microarchitectural resources identical to a single-processor implementation of the underlying microarchitecture without hardware multi-threading capability. Each logical processor in a dual-core processor (whether supporting Intel Hyper-Threading Technology or not) has its own APIC functionality, PAT, machine check architecture, debug registers and extensions. Each logical processor handles serialization instructions or self-modifying code on its own. Memory order is handled the same way as in Intel Hyper-Threading Technology.

The topology of the cache hierarchy (with respect to whether a given cache level is shared by one or more processor cores or by all logical processors in the physical package) depends on the processor implementation. Software must use the deterministic cache parameter leaf of CPUID instruction to discover the cache-sharing topology between the logical processors in a multi-threading environment.

10.8.1 Logical Processor Support

The topological composition of processor cores and logical processors in a multi-core processor can be discovered using CPUID. Within each processor core, one or more logical processors may be available.

System software must follow the requirement MP initialization sequences (see Section 10.4, "Multiple-Processor (MP) Initialization") to recognize and enable logical processors. At runtime, software can enumerate those logical processors enabled by system software to identify the topological relationships between these logical processors. (See Section 10.9.5, "Identifying Topological Relationships in an MP System").

10.8.2 Memory Type Range Registers (MTRR)

MTRR is shared between two logical processors sharing a processor core if the physical processor supports Intel Hyper-Threading Technology. MTRR is not shared between logical processors located in different cores or different physical packages.

The Intel 64 and IA-32 architectures require that all logical processors in an MP system use an identical MTRR memory map. This gives software a consistent view of memory, independent of the processor on which it is running.

See Section 13.11, "Memory Type Range Registers (MTRRs)."

10.8.3 Performance Monitoring Counters

Performance counters and their companion control MSRs are shared between two logical processors sharing a processor core if the processor core supports Intel Hyper-Threading Technology and is based on Intel NetBurst microarchitecture. They are not shared between logical processors in different cores or different physical packages. As a result, software must manage the use of these resources, based on the topology of performance monitoring resources. Performance counter interrupts, events, and precise event monitoring support can be set up and allocated on a per thread (per logical processor) basis.

See Section 21.6.4, "Performance Monitoring and Intel® Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture."

10.8.4 IA32_MISC_ENABLE MSR

Some bit fields in IA32_MISC_ENABLE MSR (MSR address 1A0H) may be shared between two logical processors sharing a processor core, or may be shared between different cores in a physical processor. See Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

10.8.5 Microcode Update Resources

Microcode update facilities are shared between two logical processors sharing a processor core if the physical package supports Intel Hyper-Threading Technology. They are not shared between logical processors in different

cores or different physical packages. Either logical processor that has access to the microcode update facility can initiate an update.

Each logical processor has its own BIOS signature MSR (IA32_BIOS_SIGN_ID at MSR address 8BH). When a logical processor performs an update for the physical processor, the IA32_BIOS_SIGN_ID MSRs for resident logical processors are updated with identical information.

All microcode update steps during processor initialization should use the same update data on all cores in all physical packages of the same stepping. Any subsequent microcode update must apply consistent update data to all cores in all physical packages of the same stepping. If the processor detects an attempt to load an older microcode update when a newer microcode update had previously been loaded, it may reject the older update to stay with the newer update.

NOTE

Some processors (prior to the introduction of Intel 64 Architecture and based on Intel NetBurst microarchitecture) do not support simultaneous loading of microcode update to the sibling logical processors in the same core. All other processors support logical processors initiating an update simultaneously. Intel recommends a common approach that the microcode loader use the sequential technique described in Section 11.11.6.3.

10.9 PROGRAMMING CONSIDERATIONS FOR HARDWARE MULTI-THREADING CAPABLE PROCESSORS

In a multi-threading environment, there may be certain hardware resources that are physically shared at some level of the hardware topology. In the multi-processor systems, typically bus and memory sub-systems are physically shared between multiple sockets. Within a hardware multi-threading capable processors, certain resources are provided for each processor core, while other resources may be provided for each logical processors (see Section 10.7, "Intel® Hyper-Threading Technology Architecture," and Section 10.8, "Multi-Core Architecture").

From a software programming perspective, control transfer of processor operation is managed at the granularity of logical processor (operating systems dispatch a runnable task by allocating an available logical processor on the platform). To manage the topology of shared resources in a multi-threading environment, it may be useful for software to understand and manage resources that are shared by more than one logical processors.

10.9.1 Hierarchical Mapping of Shared Resources

The APIC_ID value associated with each logical processor in a multi-processor system is unique (see Section 10.6, "Detecting Hardware Multi-Threading Support and Topology"). This 8-bit or 32-bit value can be decomposed into sub-fields, where each sub-field corresponds a hierarchical domain of the topological mapping of hardware resources.

The decomposition of an APIC_ID may consist of several sub fields representing the topology within a physical processor package, the higher-order bits of an APIC ID may also be used by cluster vendors to represent the topology of cluster nodes of each coherent multiprocessor systems:

- **Cluster** — Some multi-threading environments consists of multiple clusters of multi-processor systems. The CLUSTER_ID sub-field is usually supported by vendor firmware to distinguish different clusters. For non-clustered systems, CLUSTER_ID is usually 0 and system topology is reduced.
- **Package** — A physical processor package mates with a socket. A package may contain one or more software visible die. The PACKAGE_ID sub-field distinguishes different physical packages within a cluster.
- **Die** — A software-visible chip inside a package. The DIE_ID sub-field distinguishes different die within a package. If there are no software visible die, the width of this bit field is 0.
- **DieGrp** — A group of die that share certain resources.
- **Tile** — A set of cores that share certain resources. The TILE_ID sub-field distinguishes different tiles. If there are no software visible tiles, the width of this bit field is 0.

- **Module** — A set of cores that share certain resources. The MODULE_ID sub-field distinguishes different modules. If there are no software visible modules, the width of this bit field is 0.
- **Core** — Processor cores may be contained within modules, within tiles, on software-visible die, or appear directly at the package domain. The CORE_ID sub-field distinguishes processor cores. For a single-core processor, the width of this bit field is 0.
- **Logical Processor** — A processor core provides one or more logical processors sharing execution resources. The LOGICAL_PROCESSOR_ID sub-field distinguishes logical processors in a core. The width of this bit field is non-zero if a processor core provides more than one logical processors.

The LOGICAL_PROCESSOR_ID and CORE_ID sub-fields are bit-wise contiguous in the APIC_ID field (see Figure 10-5).

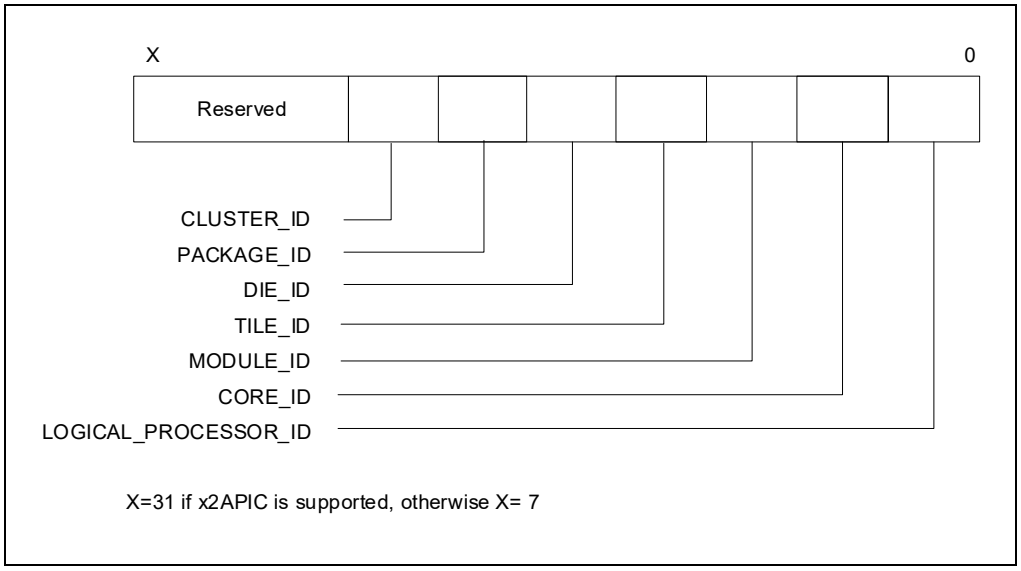


Figure 10-5. Generalized Seven-Domain Interpretation of the APIC ID

If the processor supports CPUID leaf 0BH and leaf 1FH, the 32-bit APIC ID can represent cluster plus several domains of topology within the physical processor package. The exact number of hierarchical domains within a physical processor package must be enumerated through CPUID leaf 0BH and leaf 1FH. Common processor families may employ a topology similar to that represented by the 8-bit Initial APIC ID. In general, CPUID leaf 0BH and leaf 1FH can support a topology enumeration algorithm that decompose a 32-bit APIC ID into more than four sub-fields (see Figure 10-6).

NOTE

CPUID leaf 0BH and leaf 1FH can have differences in the number of domain types reported (CPUID leaf 1FH defines additional domain types). If the processor supports CPUID leaf 1FH, usage of this leaf is preferred over leaf 0BH. CPUID leaf 0BH is available for legacy compatibility going forward.

The width of each sub-field depends on hardware and software configurations. Field widths can be determined at runtime using the algorithm discussed below (Example 10-16 through Example 10-21).

Figure 7-6 depicts the relationships of three of the hierarchical sub-fields in a hypothetical MP system. The value of valid APIC_IDs need not be contiguous across package boundary or core boundaries.

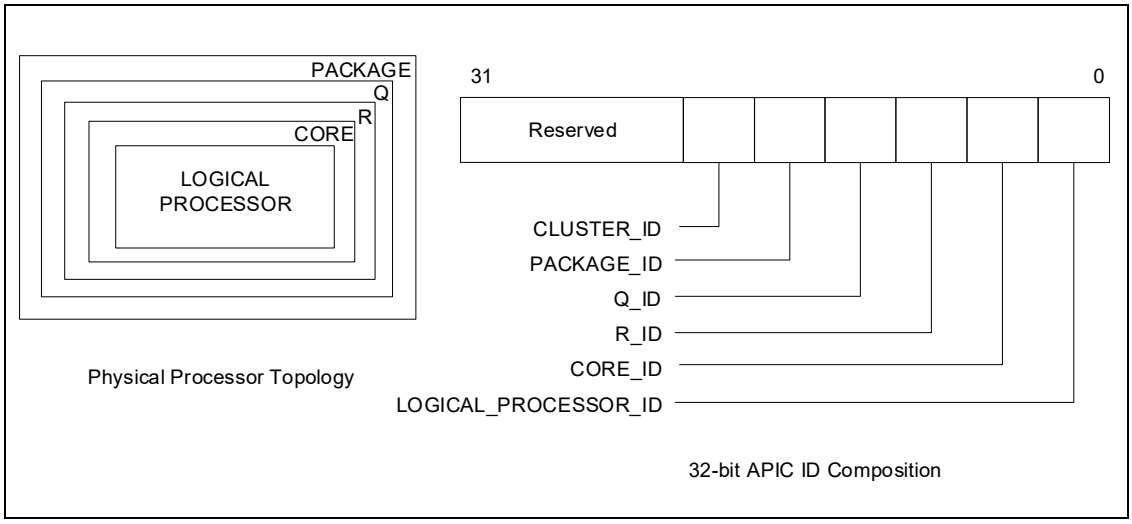


Figure 10-6. Conceptual Six-Domain Topology and 32-bit APIC ID Composition

10.9.2 Hierarchical Mapping of CPUID Extended Topology Leaf

CPUID leaf 0BH and leaf 1FH provide enumeration parameters for software to identify each hierarchy of the processor topology in a deterministic manner. Each hierarchical domain of the topology starting from the Logical Processor domain is represented numerically by a sub-leaf index within the CPUID 0BH leaf and 1FH leaf. Each domain of the topology is mapped to a sub-field in the APIC ID, following the general relationship depicted in Figure 10-6. This mechanism allows software to query the exact number of domains within a physical processor package and the bit-width of each sub-field of x2APIC ID directly. For example,

- Starting from sub-leaf index 0 and incrementing ECX until CPUID.(EAX=0BH or 1FH, ECX=N):ECX[15:8] returns an invalid "domain type" encoding. The number of domains within the physical processor package is "N" (excluding PACKAGE). Using Figure 10-6 as an example, CPUID.(EAX=0BH or 1FH, ECX=4):ECX[15:8] will report 00H, indicating sub leaf 04H is invalid. This is also depicted by a pseudo code example:

Example 10-16. Number of Domains Below the Physical Processor Package

```
Word NumberOfDomainsBelowPackage = 0;
DWord Subleaf = 0;

EAX = 0BH or 1FH; // query each sub leaf of CPUID leaf 0BH or 1FH; CPUID leaf 1FH is preferred over leaf 0BH if available.
ECX = Subleaf;
CPUID;
while(EBX != 0)  // Enumerate until EBX reports 0
{
     if(EAX[4:0] != 0)    // A Shift Value of 0 indicates this domain does not exist.
                          // (Such as no SMT_ID, which is required entry at sub-leaf 0.)
    {
         NumberOfDomainsBelowPackage++;
    }
    Subleaf++;
    EAX = 0BH or 1FH;
    ECX = Subleaf;
    CPUID;
}
// NumberOfDomainsBelowPackage contains the absolute number of domains that exist below package.
N = Subleaf;   // Sub-leaf supplies the number of entries CPUID will return.
```

- Sub-leaf index 0 (ECX= 0 as input) provides enumeration parameters to extract the LOGICAL_PROCESSOR_ID sub-field of x2APIC ID. If EAX = 0BH or 1FH, and ECX =0 is specified as input when executing CPUID, CPUID.(EAX=0BH or 1FH, ECX=0):EAX[4:0] reports a value (a right-shift count) that allow software to extract part of x2APIC ID to distinguish the next higher topological entities above the LOGICAL_PROCESSOR_ID domain. This value also corresponds to the bit-width of the sub-field of x2APIC ID corresponding the hierarchical domain with sub-leaf index 0.
- For each subsequent higher sub-leaf index m, CPUID.(EAX=0BH or 1FH, ECX=m):EAX[4:0] reports the right-shift count that will allow software to extract part of x2APIC ID to distinguish higher-domain topological entities. This means the right-shift value at of sub-leaf m, corresponds to the least significant (m+1) sub-fields of the 32-bit x2APIC ID.

Example 10-17. BitWidth Determination of x2APIC ID Sub-fields

```
For m = 0, m < N, m ++;
{    cumulative_width[m] = CPUID.(EAX=0BH or 1FH, ECX= m): EAX[4:0]; }
BitWidth[0] = cumulative_width[0];
For m = 1, m < N, m ++;
    BitWidth[m] = cumulative_width[m] - cumulative_width[m-1];
```

NOTE

CPUID leaf 1FH is a preferred superset to leaf 0BH. Leaf 1FH defines additional domain types, and it must be parsed by an algorithm that can handle the addition of future domain types.

Previously, only the following encoding of hierarchical domain types were defined: 0 (invalid), 1 (logical processor), and 2 (core). With the additional hierarchical domain types available (see Section 10.9.1, "Hierarchical Mapping of Shared Resources," and Figure 10-5, "Generalized Seven-Domain Interpretation of the APIC ID") software must not assume any "domain type" encoding value to be related to any sub-leaf index, except sub-leaf 0.

Example 10-18. Support Routines for Identifying Package, Die, Core, and Logical Processors from 32-bit x2APIC ID

a. Derive the extraction bitmask for logical processors in a processor core and associated mask offset for different cores.

```
//
// This example shows how to enumerate CPU topology domain types (domain types may or may not be known/supported by the
software)
//
// Below is the list of sample domain types used in the example.
// Refer to the CPUID Leaf 1FH definition for the actual domain type numbers: "V2 Extended Topology Enumeration Leaf (Initial EAX
Value = 1FH, ECX ≥ 0)" .
//
// LOGICAL PROCESSOR
// CORE
// MODULE
// TILE
// DIE
// PACKAGE
//
// The example shows how to identify and derive the extraction bitmask for the domains with identify type
LOGICAL_PROCESSOR_ID/CORE_ID/DIE_ID/PACKAGE_ID
//

int DeriveLogical_Processor_Mask_Offsets (void)
{
```

```
    IF (!HWMTSupported()) return -1;
    execute cpuid with EAX = 0BH or 1FH, ECX = 0;
    IF (returned domain type encoding in EXC[15:8] does not match LOGICAL_PROCESSOR_ID) return -1;
    Mask_Logical_Processor_shift = EAX[4:0];        //# bits shift right of APIC ID to distinguish different cores, note this can be a shift
                                                    // of zero if there is only one logical processor per core.
    Logical Processor Mask =~( (-1) << Mask_Logical_Processor_shift);      //shift left to derive extraction bitmask for
                                                                           // LOGICAL_PROCESSOR_ID
    return 0;
}
```

b. Derive the extraction bitmask for processor cores in a physical processor package and associated mask offset for different packages.

```
int DeriveCore_Mask_Offsets (void)
{
    IF (!HWMTSupported()) return -1;
    execute cpuid with EAX = 0BH or 1FH, ECX = 0;
    WHILE( ECX[15:8] ) {                            //domain type encoding is valid
        Mask_last_known_shift = EAX[4:0]
        IF (returned domain type encoding in ECX[15:8] matches CORE) {
            Mask_Core_shift = EAX[4:0];
        }
        ELSE IF (returned domain type encoding in ECX[15:8] matches DIE {
            Mask_Die_shift = EAX[4:0];
        }
        //
        // Keep enumerating. Check if the next domain is the desired domain and if not, keep enumerating until you reach a known
        // domain or the invalid domain ("0" domain type). If there are more domains between DIE and PACKAGE, the unknown
        // domains will be ignored and treated as an extension of the last known domain (i.e., DIE in this case).
        //

        ECX++;
        execute cpuid with EAX = 0BH or 1FH;
    }

    COREPlusLogical_Processor_MASK = ~( (-1) << Mask_Core_shift);
    DIEPlusCORE_MASK = ~( (-1) << Mask_Die_shift);

    //
    // Treat domains between DIE and physical package as an extension of DIE for software choosing not to implement or recognize
    // these unknown domains.
    //

    CORE_MASK = COREPlusLogical_Processor_MASK ^ Logical Processor Mask;
    DIE_MASK = DIEPlusCORE_MASK ^ COREPlusLogical_Processor_MASK;
    PACKAGE_MASK = (-1) << Mask_last_known_shift;

    return -1;

}
```

10.9.3 Hierarchical ID of Logical Processors in an MP System

For Intel 64 and IA-32 processors, system hardware establishes an 8-bit initial APIC ID (or 32-bit APIC ID if the processor supports CPUID leaf 0BH) that is unique for each logical processor following power-up or RESET (see Section 10.6.1). Each logical processor on the system is allocated an initial APIC ID. BIOS may implement features that tell the OS to support less than the total number of logical processors on the system bus. Those logical processors that are not available to applications at runtime are halted during the OS boot process. As a result, the number valid local APIC_IDs that can be queried by affinitizing-current-thread-context (See Example 10-23) is limited to the number of logical processors enabled at runtime by the OS boot process.

Table 10-2 shows an example of the 8-bit APIC IDs that are initially reported for logical processors in a system with four Intel Xeon MP processors that support Intel Hyper-Threading Technology (a total of 8 logical processors, each physical package has two processor cores and supports Intel Hyper-Threading Technology). Of the two logical processors within a Intel Xeon processor MP, logical processor 0 is designated the primary logical processor and logical processor 1 as the secondary logical processor.

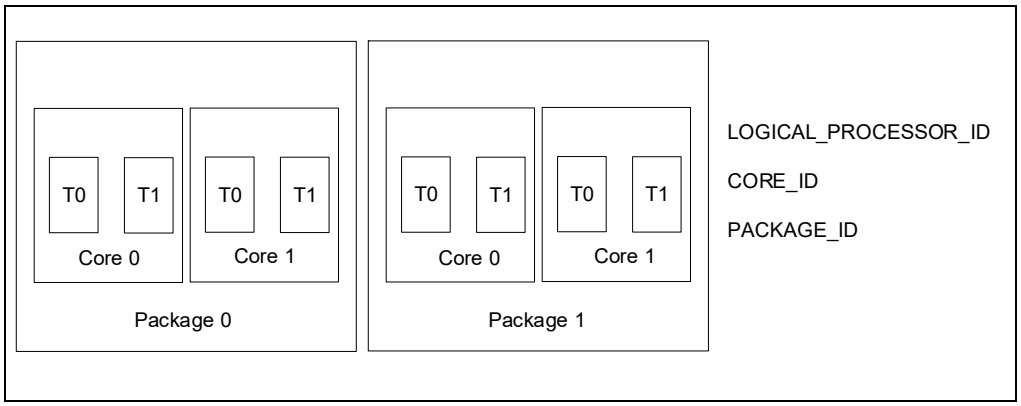


Figure 10-7. Topological Relationships Between Hierarchical IDs in a Hypothetical MP Platform

Table 10-2. Initial APIC IDs for the Logical Processors in a System that has Four Intel Xeon MP Processors Supporting Intel Hyper-Threading Technology[1]

Initial APIC ID	PACKAGE_ID	CORE_ID	LOGICAL_PROCESSOR_ID
0H	0H	0H	0H
1H	0H	0H	1H
2H	1H	0H	0H
3H	1H	0H	1H
4H	2H	0H	0H
5H	2H	0H	1H
6H	3H	0H	0H
7H	3H	0H	1H

NOTE:

1. Because information on the number of processor cores in a physical package was not available in early single-core processors supporting Intel Hyper-Threading Technology, the CORE_ID can be treated as 0.

Table 10-3 shows the initial APIC IDs for a hypothetical situation with a dual processor system. Each physical package providing two processor cores, and each processor core also supporting Intel Hyper-Threading Technology.

Table 10-3. Initial APIC IDs for the Logical Processors in a System that has Two Physical Processors Supporting Dual-Core and Intel Hyper-Threading Technology

Initial APIC ID	PACKAGE_ID	CORE_ID	LOGICAL_PROCESSOR_ID
0H	0H	0H	0H
1H	0H	0H	1H
2H	0H	1H	0H
3H	0H	1H	1H
4H	1H	0H	0H
5H	1H	0H	1H
6H	1H	1H	0H
7H	1H	1H	1H

10.9.3.1 Hierarchical ID of Logical Processors with x2APIC ID

Table 10-4 shows an example of possible x2APIC ID assignments for a dual processor system that support x2APIC. Each physical package providing four processor cores, and each processor core also supporting Intel Hyper-Threading Technology. Note that the x2APIC ID need not be contiguous in the system.

Table 10-4. Example of Possible x2APIC ID Assignment in a System that has Two Physical Processors Supporting x2APIC and Intel Hyper-Threading Technology

x2APIC ID	PACKAGE_ID	CORE_ID	LOGICAL_PROCESSOR_ID
0H	0H	0H	0H
1H	0H	0H	1H
2H	0H	1H	0H
3H	0H	1H	1H
4H	0H	2H	0H
5H	0H	2H	1H
6H	0H	3H	0H
7H	0H	3H	1H
10H	1H	0H	0H
11H	1H	0H	1H
12H	1H	1H	0H
13H	1H	1H	1H
14H	1H	2H	0H
15H	1H	2H	1H
16H	1H	3H	0H
17H	1H	3H	1H

10.9.4 Algorithm for Three-Domain Mappings of APIC_ID

Software can gather the initial APIC_IDs for each logical processor supported by the operating system at runtime[1] and extract identifiers corresponding to the three domains of sharing topology (package, core, and logical processor). The three-domain algorithms below focus on a non-clustered MP system for simplicity. They do not assume APIC IDs are contiguous or that all logical processors on the platform are enabled.

Intel supports multi-threading systems where all physical processors report identical values in CPUID leaf 0BH, CPUID.1:EBX[23:16]), CPUID.4[2]:EAX[31:26], and CPUID.4[3]:EAX[25:14]. The algorithms below assume the target system has symmetry across physical package boundaries with respect to the number of logical processors per package, number of cores per package, and cache topology within a package.

Software can choose to assume three-domain hierarchy if it was developed to understand only three domains. However, software implementation needs to ensure it does not break if it runs on systems that have more domains in the hierarchy even if it does not recognize them.

The extraction algorithm (for three-domain mappings from an APIC ID) uses the general procedure depicted in Example 10-19, and is supplemented by more detailed descriptions on the derivation of topology enumeration parameters for extraction bit masks:

1. Detect hardware multi-threading support in the processor.
2. Derive a set of bit masks that can extract the sub ID of each hierarchical domain of the topology. The algorithm to derive extraction bit masks for LOGICAL_PROCESSOR_ID/CORE_ID/PACKAGE_ID differs based on APIC ID is 32-bit (see step 3 below) or 8-bit (see step 4 below).
3. If the processor supports CPUID leaf 0BH, each APIC ID contains a 32-bit value, the topology enumeration parameters needed to derive three-domain extraction bit masks are:
   a. Query the right-shift value for the LOGICAL_PROCESSOR_ID domain of the topology using CPUID leaf 0BH with ECX =0H as input. The number of bits to shift-right on x2APIC ID (EAX[4:0]) can distinguish different higher-domain entities above logical processor in the same physical package. This is also the width of the bit mask to extract the LOGICAL_PROCESSOR_ID. The shift value may be 0 and enumerate no logical processor bit mask to create. A platform where cores only have one logical processor are not required to enumerate a separate bit layout for logical processor, and the lowest bits may only identify the core (where core and logical processor are then synonymous).
   b. Enumerate until the desired domain is found (i.e., processor cores). Determine if the next domain is the expected domain. If the next domain is not known to the software, keep enumerating until the next known or the last domain. Software should use the previous domain before this to represent the last previously known domain (i.e., processor cores). If the software does not recognize or implement certain hierarchical domains, it should assume these unknown domains as an extension of the last known domain.
   c. Query CPUID leaf 0BH for the amount of bit shift to distinguish next higher-domain entities (e.g., physical processor packages) in the system. This describes an explicit three-domain-topology situation for commonly available processors. Consult Example 10-17 to adapt to situations beyond a three-domain topology of a physical processor. The width of the extraction bit mask can be used to derive the cumulative extraction bitmask to extract the sub IDs of logical processors (including different processor cores) in the same physical package. The extraction bit mask to distinguish merely different processor cores can be derived by xor'ing the logical processor extraction bit mask from the cumulative extraction bit mask.
   d. Query the 32-bit x2APIC ID for the logical processor where the current thread is executing.
   e. Derive the extraction bit masks corresponding to LOGICAL_PROCESSOR_ID, CORE_ID, and PACKAGE_ID, starting from LOGICAL_PROCESSOR_ID.
   f. Apply each extraction bit mask to the 32-bit x2APIC ID to extract sub-field IDs.

---

1. As noted in Section 10.6 and Section 10.9.3, the number of logical processors supported by the OS at runtime may be less than the total number logical processors available in the platform hardware.
2. Maximum number of addressable ID for processor cores in a physical processor is obtained by executing CPUID with EAX=4 and a valid ECX index. The ECX index starts at 0.
3. Maximum number addressable ID for processor cores sharing the target cache level is obtained by executing CPUID with EAX = 4 and the ECX index corresponding to the target cache level.

4. If the processor does not support CPUID leaf 0BH, each initial APIC ID contains an 8-bit value, the topology enumeration parameters needed to derive extraction bit masks are:
   a. Query the size of address space for sub IDs that can accommodate logical processors in a physical processor package. This size parameters (CPUID.1:EBX[23:16]) can be used to derive the width of an extraction bitmask to enumerate the sub IDs of different logical processors in the same physical package.
   b. Query the size of address space for sub IDs that can accommodate processor cores in a physical processor package. This size parameters can be used to derive the width of an extraction bitmask to enumerate the sub IDs of processor cores in the same physical package.
   c. Query the 8-bit initial APIC ID for the logical processor where the current thread is executing.
   d. Derive the extraction bit masks using respective address sizes corresponding to LOGICAL_PROCESSOR_ID, CORE_ID, and PACKAGE_ID, starting from LOGICAL_PROCESSOR_ID.
   e. Apply each extraction bit mask to the 8-bit initial APIC ID to extract sub-field IDs.

Example 10-19. Support Routines for Detecting Hardware Multi-Threading and Identifying the Relationships Between Package, Core, and Logical Processors

1. Detect support for Hardware Multi-Threading Support in a processor.

```
//  Returns a non-zero value if CPUID reports the presence of hardware multi-threading
//  support in the physical package where the current logical processor is located.
//  This does not guarantee BIOS or OS will enable all logical processors in the physical
//  package and make them available to applications.
//  Returns zero if hardware multi-threading is not present.

#define HWMT_BIT 10000000H

unsigned int HWMTSupported(void)
{
     // ensure cpuid instruction is supported
          execute cpuid with eax = 0 to get vendor string
          execute cpuid with eax = 1 to get feature flag and signature


     // Check to see if this a Genuine Intel Processor

     if (vendor string EQ GenuineIntel) {
          return (feature_flag_edx & HWMT_BIT); // bit 28
     }
     return 0;
}
```

Example 10-20. Support Routines for Identifying Package, Core, and Logical Processors from 32-bit x2APIC ID

a. Derive the extraction bitmask for logical processors in a processor core and associated mask offset for different cores.

```
int DeriveLogical_Processor_Mask_Offsets (void)
{
     if (!HWMTSupported()) return -1;
     execute cpuid with eax = 11, ECX = 0;
     If (returned domain type encoding in ECX[15:8] does not match logical processor) return -1;
     Mask_Logical_Processor_shift = EAX[4:0];       // # bits shift right of APIC ID to distinguish different cores, note this can be a shift
                                                    // of zero if there is only one logical processor per core.
     Logical Processor Mask = ~( (-1) << Mask_Logical_Processor_shift); // shift left to derive extraction bitmask for
                                                                        // LOGICAL_PROCESSOR_ID
```

```
    return 0;
}
```

b. Derive the extraction bitmask for processor cores in a physical processor package and associated mask offset for different packages.

```
int DeriveCore_Mask_Offsets (void)
{
    if (!HWMTSupported()) return -1;
execute cpuid with eax = 11, ECX = 0;
    while( ECX[15:8] ) {            // domain type encoding is valid
        Mask_Core_shift = EAX[4:0];      // needed to distinguish different physical packages
        ECX ++;
        execute cpuid with eax = 11;
   }
    COREPlusLogical_Processor_MASK = ~( (-1) << Mask_Core_shift);
    // treat domains between core and physical package as a core for software choosing not to implement or recognize
    // these unknown domains
    CORE_MASK = COREPlusLogical_Processor_MASK ^ Logical Processor Mask;
    PACKAGE_MASK = (-1) << Mask_Core_shift;
    return -1;
}
```

c. Query the x2APIC ID of a logical processor.

APIC_IDs for each logical processor.

```
unsigned char Getx2APIC_ID (void)
{
    unsigned reg_edx = 0;
    execute cpuid with eax = 11, ECX = 0
    store returned value of edx
    return (unsigned) (reg_edx) ;
}
```

Example 10-21. Support Routines for Identifying Package, Core, and Logical Processors from 8-bit Initial APIC ID

a. Find the size of address space for logical processors in a physical processor package.

```
#define NUM_LOGICAL_BITS 00FF0000H
// Use the mask above and CPUID.1.EBX[23:16] to obtain the max number of addressable IDs
// for logical processors in a physical package,

//Returns the size of address space of logical processors in a physical processor package;
// Software should not assume the value to be a power of 2.

unsigned char MaxLPIDsPerPackage(void)
{
    if (!HWMTSupported()) return 1;
execute cpuid with eax = 1
    store returned value of ebx
    return (unsigned char) ((reg_ebx & NUM_LOGICAL_BITS) >> 16);
}
```

b. Find the size of address space for processor cores in a physical processor package.

```
// Returns the max number of addressable IDs for processor cores in a physical processor package;
// Software should not assume cpuid reports this value to be a power of 2.

unsigned MaxCoreIDsPerPackage(void)
{
    if (!HWMTSupported()) return (unsigned char) 1;
    if cpuid supports leaf number 4
    { // we can retrieve multi-core topology info using leaf 4
        execute cpuid with eax = 4, ecx = 0
        store returned value of eax
        return (unsigned) ((reg_eax >> 26) +1);
    }
    else // must be a single-core processor
    return 1;
}
```

c. Query the initial APIC ID of a logical processor.

```
#define INITIAL_APIC_ID_BITS FF000000H // CPUID.1.EBX[31:24] initial APIC ID

// Returns the 8-bit unique initial APIC ID for the processor running the code.
// Software can use OS services to affinitize the current thread to each logical processor
// available under the OS to gather the initial APIC_IDs for each logical processor.

unsigned GetInitAPIC_ID (void)
{
    unsigned int reg_ebx = 0;
    execute cpuid with eax = 1
    store returned value of ebx
    return (unsigned) ((reg_ebx & INITIAL_APIC_ID_BITS) >> 24;
}
```

d. Find the width of an extraction bitmask from the maximum count of the bit-field (address size).

```
// Returns the mask bit width of a bit field from the maximum count that bit field can represent.
// This algorithm does not assume 'address size' to have a value equal to power of 2.
// Address size for LOGICAL_PROCESSOR_ID can be calculated from MaxLPIDsPerPackage()/MaxCoreIDsPerPackage()
// Then use the routine below to derive the corresponding width of logical processor extraction bitmask
// Address size for CORE_ID is MaxCoreIDsPerPackage(),
// Derive the bitwidth for CORE extraction mask similarly

unsigned FindMaskWidth(Unsigned Max_Count)
{unsigned int mask_width, cnt = Max_Count;
    __asm {
            mov eax, cnt
            mov ecx, 0
            mov mask_width, ecx
            dec eax
            bsr cx, ax
            jz next
            inc cx
            mov  mask_width, ecx
            next:
            mov eax, mask_width
```

```
        }
    return mask_width;
}
```

e. Extract a sub ID from an 8-bit full ID, using address size of the sub ID and shift count.

```
// The routine below can extract LOGICAL_PROCESSOR_ID, CORE_ID, and PACKAGE_ID respectively from the init APIC_ID
// To extract LOGICAL_PROCESSOR_ID, MaxSubIDvalue is set to the address size of LOGICAL_PROCESSOR_ID, Shift_Count = 0
// To extract CORE_ID, MaxSubIDvalue is the address size of CORE_ID, Shift_Count is width of logical processor extraction bitmask.
// Returns the value of the sub ID, this is not a zero-based value

Unsigned char GetSubID(unsigned char Full_ID, unsigned char MaxSubIDvalue, unsigned char Shift_Count)
{
    MaskWidth = FindMaskWidth(MaxSubIDValue);
    MaskBits = ((uchar) (FFH << Shift_Count)) ^ ((uchar) (FFH << Shift_Count + MaskWidth)) ;
    SubID = Full_ID & MaskBits;
    Return SubID;
}
```

Software must not assume local APIC_ID values in an MP system are consecutive. Non-consecutive local APIC_IDs may be the result of hardware configurations or debug features implemented in the BIOS or OS.

An identifier for each hierarchical domain can be extracted from an 8-bit APIC_ID using the support routines illustrated in Example 10-21. The appropriate bit mask and shift value to construct the appropriate bit mask for each domain must be determined dynamically at runtime.

10.9.5 Identifying Topological Relationships in an MP System

To detect the number of physical packages, processor cores, or other topological relationships in a MP system, the following procedures are recommended:

- Extract the three-domain identifiers from the APIC ID of each logical processor enabled by system software. The sequence is as follows (see the pseudo code shown in Example 10-22 and support routines shown in Example 10-19):
    - The extraction start from the right-most bit field, corresponding to LOGICAL_PROCESSOR_ID, the innermost hierarchy in a three-domain topology (See Figure 10-7). For the right-most bit field, the shift value of the working mask is zero. The width of the bit field is determined dynamically using the maximum number of logical processor per core, which can be derived from information provided from CPUID.
    - To extract the next bit-field, the shift value of the working mask is determined from the width of the bit mask of the previous step. The width of the bit field is determined dynamically using the maximum number of cores per package.
    - To extract the remaining bit-field, the shift value of the working mask is determined from the maximum number of logical processor per package. So the remaining bits in the APIC ID (excluding those bits already extracted in the two previous steps) are extracted as the third identifier. This applies to a non-clustered MP system, or if there is no need to distinguish between PACKAGE_ID and CLUSTER_ID.

      If there is need to distinguish between PACKAGE_ID and CLUSTER_ID, PACKAGE_ID can be extracted using an algorithm similar to the extraction of CORE_ID, assuming the number of physical packages in each node of a clustered system is symmetric.
- Assemble the three-domain identifiers of LOGICAL_PROCESSOR_ID, CORE_ID, PACKAGE_IDs into arrays for each enabled logical processor. This is shown in Example 10-23a.
- To detect the number of physical packages: use PACKAGE_ID to identify those logical processors that reside in the same physical package. This is shown in Example 10-23b. This example also depicts a technique to construct a mask to represent the logical processors that reside in the same package.

- To detect the number of processor cores: use CORE_ID to identify those logical processors that reside in the same core. This is shown in Example 10-23. This example also depicts a technique to construct a mask to represent the logical processors that reside in the same core.

In Example 10-22, the numerical ID value can be obtained from the value extracted with the mask by shifting it right by shift count. Algorithms below do not shift the value. The assumption is that the SubID values can be compared for equivalence without the need to shift.

Example 10-22. Pseudo Code Depicting Three-Domain Extraction Algorithm

```
For Each local_APIC_ID{
    // Calculate Logical Processor Mask, the bit mask pattern to extract LOGICAL_PROCESSOR_ID,
    // Logical Processor Mask is determined using topology enumertaion parameters
    // from CPUID leaf 0BH (Example 10-20);
    // otherwise, Logical Processor Mask is determined using CPUID leaf 01H and leaf 04H (Example 10-21).
    // This algorithm assumes there is symmetry across core boundary, i.e., each core within a
    //  package has the same number of logical processors
    // LOGICAL_PROCESSOR_ID always starts from bit 0, corresponding to the right-most bit-field
    LOGICAL_PROCESSOR_ID = APIC_ID & Logical Processor Mask;

// Extract CORE_ID:
    // Core Mask is determined in Example 10-20 or Example 10-21
    CORE_ID = (APIC_ID & Core Mask);

    // Extract PACKAGE_ID:
    // Assume single cluster.
    // Shift out the mask width for maximum logical processors per package
    // Package Mask is determined in Example 10-20 or Example 10-21
    PACKAGE_ID = (APIC_ID & Package Mask) ;
}
```

Example 10-23. Compute the Number of Packages, Cores, and Processor Relationships in a MP System

a) Assemble lists of PACKAGE_ID, CORE_ID, and LOGICAL_PROCESSOR_ID of each enabled logical processors

```
// The BIOS and/or OS may limit the number of logical processors available to applications after system boot.
// The below algorithm will compute topology for the processors visible to the thread that is computing it.

// Extract the 3-domains of IDs on every processor.
// SystemAffinity is a bitmask of all the processors started by the OS. Use OS specific APIs to obtain it.
// ThreadAffinityMask is used to affinitize the topology enumeration thread to each processor using OS specific APIs.
// Allocate per processor arrays to store the Package_ID, Core_ID, and LOGICAL_PROCESSOR_ID for every started processor.

    ThreadAffinityMask = 1;
   ProcessorNum = 0;
    while (ThreadAffinityMask ≠ 0 && ThreadAffinityMask <= SystemAffinity) {
        // Check to make sure we can utilize this processor first.
        if (ThreadAffinityMask & SystemAffinity){
            Set thread to run on the processor specified in ThreadAffinityMask
            Wait if necessary and ensure thread is running on specified processor

            APIC_ID = GetAPIC_ID(); // 32 bit ID in Example 10-20 or 8-bit ID in Example 10-21
            Extract the Package_ID, Core_ID, and LOGICAL_PROCESSOR_ID as explained in three domain extraction
                algorithm of Example 10-22
            PackageID[ProcessorNUM] = PACKAGE_ID;
            CoreID[ProcessorNum] = CORE_ID;
```

```
            LOGICAL_PROCESSOR_ID[ProcessorNum] = LOGICAL_PROCESSOR_ID;
            ProcessorNum++;
        }
        ThreadAffinityMask <<= 1;
    }
    NumStartedLPs = ProcessorNum;
```

b) Using the list of PACKAGE_ID to count the number of physical packages in a MP system and construct, for each package, a multi-bit mask corresponding to those logical processors residing in the same package.

```
    // Compute the number of packages by counting the number of processors with unique PACKAGE_IDs in the PackageID array.
    // Compute the mask of processors in each package.

    // PackageIDBucket is an array of unique PACKAGE_ID values. Allocate an array of NumStartedLPs count of entries in this array.
    // PackageProcessorMask is a corresponding array of the bit mask of processors belonging to the same package, these are
    // processors with the same PACKAGE_ID.
    // The algorithm below assumes there is symmetry across package boundary if more than one socket is populated in an MP
    //system.
    // Bucket Package IDs and compute processor mask for every package.

    PackageNum = 1;
    PackageIDBucket[0] = PackageID[0];
    ProcessorMask = 1;
    PackageProcessorMask[0] = ProcessorMask;
    For (ProcessorNum = 1; ProcessorNum < NumStartedLPs; ProcessorNum++) {
        ProcessorMask << = 1;
        For (i=0; i < PackageNum; i++) {
            // we may be comparing bit-fields of logical processors residing in different
            // packages, the code below assume package symmetry
            If (PackageID[ProcessorNum] = PackageIDBucket[i]) {
                PackageProcessorMask[i] |= ProcessorMask;
                Break; // found in existing bucket, skip to next iteration
            }
        }
        if (i =PackageNum) {
            //PACKAGE_ID did not match any bucket, start new bucket
            PackageIDBucket[i] = PackageID[ProcessorNum];
            PackageProcessorMask[i] = ProcessorMask;
            PackageNum++;
        }
    }
    // PackageNum has the number of Packages started in OS
    // PackageProcessorMask[] array has the processor set of each package
```

c) Using the list of CORE_ID to count the number of cores in a MP system and construct, for each core, a multi-bit mask corresponding to those logical processors residing in the same core.

Processors in the same core can be determined by bucketing the processors with the same PACKAGE_ID and CORE_ID. Note that code below can BIT OR the values of PACKGE and CORE ID because they have not been shifted right.
The algorithm below assumes there is symmetry across package boundary if more than one socket is populated in an MP system.

```
//Bucketing PACKAGE and CORE IDs and computing processor mask for every core
    CoreNum = 1;
    CoreIDBucket[0] = PackageID[0] | CoreID[0];
    ProcessorMask = 1;
```

```
CoreProcessorMask[0] = ProcessorMask;
For (ProcessorNum = 1; ProcessorNum < NumStartedLPs; ProcessorNum++) {
      ProcessorMask << = 1;
      For (i=0; i < CoreNum; i++) {
            // we may be comparing bit-fields of logical processors residing in different
            // packages, the code below assume package symmetry
            If ((PackageID[ProcessorNum] | CoreID[ProcessorNum]) = CoreIDBucket[i]) {
                  CoreProcessorMask[i] |= ProcessorMask;
                  Break; // found in existing bucket, skip to next iteration
            }
      }
      if (i = CoreNum) {
            //Did not match any bucket, start new bucket
            CoreIDBucket[i] = PackageID[ProcessorNum] | CoreID[ProcessorNum];
            CoreProcessorMask[i] = ProcessorMask;
            CoreNum++;
      }
}
// CoreNum has the number of cores started in the OS
// CoreProcessorMask[] array has the processor set of each core
```

Other processor relationships such as processor mask of sibling cores can be computed from set operations of the PackageProcessorMask[] and CoreProcessorMask[].

The algorithm shown above can be adapted to work with earlier generations of single-core IA-32 processors that support Intel Hyper-Threading Technology and in situations that the deterministic cache parameter leaf is not supported (provided CPUID supports initial APIC ID). A reference code example is available (see Intel® 64 Architecture Processor Topology Enumeration Technical Paper).

10.10 MANAGEMENT OF IDLE AND BLOCKED CONDITIONS

When a logical processor in an MP system (including multi-core processor or processors supporting Intel Hyper-Threading Technology) is idle (no work to do) or blocked (on a lock or semaphore), additional management of the core execution engine resource can be accomplished by using the HLT (halt), PAUSE, or the MONITOR/MWAIT instructions.

10.10.1 HLT Instruction

The HLT instruction stops the execution of the logical processor on which it is executed and places it in a halted state until further notice (see the description of the HLT instruction in Chapter 3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A). When a logical processor is halted, active logical processors continue to have full access to the shared resources within the physical package. Here shared resources that were being used by the halted logical processor become available to active logical processors, allowing them to execute at greater efficiency. When the halted logical processor resumes execution, shared resources are again shared among all active logical processors. (See Section 10.10.6.3, "Halt Idle Logical Processors," for more information about using the HLT instruction with processors supporting Intel Hyper-Threading Technology.)

10.10.2 PAUSE Instruction

The PAUSE instruction can improves the performance of processors supporting Intel Hyper-Threading Technology when executing "spin-wait loops" and other routines where one thread is accessing a shared lock or semaphore in a tight polling loop. When executing a spin-wait loop, the processor can suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation and flushes the core processor's pipeline. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses

this hint to avoid the memory order violation and prevent the pipeline flush. In addition, the PAUSE instruction de-pipelines the spin-wait loop to prevent it from consuming execution resources excessively and consume power needlessly. (See Section 10.10.6.1, "Use the PAUSE Instruction in Spin-Wait Loops," for more information about using the PAUSE instruction with IA-32 processors supporting Intel Hyper-Threading Technology.)

10.10.3 Detecting Support MONITOR/MWAIT Instruction

Streaming SIMD Extensions 3 introduced two instructions (MONITOR and MWAIT) to help multithreaded software improve thread synchronization. In the initial implementation, MONITOR and MWAIT are available to software at ring 0. The instructions are conditionally available at levels greater than 0. Use the following steps to detect the availability of MONITOR and MWAIT:

- Use CPUID to query the MONITOR bit (CPUID.1.ECX[3] = 1).
- If CPUID indicates support, execute MONITOR inside a TRY/EXCEPT exception handler and trap for an exception. If an exception occurs, MONITOR and MWAIT are not supported at a privilege level greater than 0. See Example 10-24.

Example 10-24. Verifying MONITOR/MWAIT Support

```
boolean MONITOR_MWAIT_works = TRUE;
try {
  _asm {
    xor ecx, ecx
    xor edx, edx
    mov eax, MemArea
    monitor
    }
     // Use monitor
} except (UNWIND) {
     // if we get here, MONITOR/MWAIT is not supported
    MONITOR_MWAIT_works = FALSE;
}
```

10.10.4 MONITOR/MWAIT Instruction

Operating systems usually implement idle loops to handle thread synchronization. In a typical idle-loop scenario, there could be several "busy loops" and they would use a set of memory locations. An impacted processor waits in a loop and poll a memory location to determine if there is available work to execute. The posting of work is typically a write to memory (the work-queue of the waiting processor). The time for initiating a work request and getting it scheduled is on the order of a few bus cycles.

From a resource sharing perspective (logical processors sharing execution resources), use of the HLT instruction in an OS idle loop is desirable but has implications. Executing the HLT instruction on a idle logical processor puts the targeted processor in a non-execution state. This requires another processor (when posting work for the halted logical processor) to wake up the halted processor using an inter-processor interrupt. The posting and servicing of such an interrupt introduces a delay in the servicing of new work requests.

In a shared memory configuration, exits from busy loops usually occur because of a state change applicable to a specific memory location; such a change tends to be triggered by writes to the memory location by another agent (typically a processor).

MONITOR/MWAIT complement the use of HLT and PAUSE to allow for efficient partitioning and un-partitioning of shared resources among logical processors sharing physical resources. MONITOR sets up an effective address range that is monitored for write-to-memory activities; MWAIT places the processor in an optimized state (this may vary between different implementations) until a write to the monitored address range occurs.

In the initial implementation of MONITOR and MWAIT, they are available at CPL = 0 only.

Both instructions rely on the state of the processor's monitor hardware. The monitor hardware can be either armed (by executing the MONITOR instruction) or triggered (due to a variety of events, including a store to the monitored memory region). If upon execution of MWAIT, monitor hardware is in a triggered state: MWAIT behaves as a NOP and execution continues at the next instruction in the execution stream. The state of monitor hardware is not architecturally visible except through the behavior of MWAIT.

Multiple events other than a write to the triggering address range can cause a processor that executed MWAIT to wake up. These include events that would lead to voluntary or involuntary context switches, such as:

- External interrupts, including NMI, SMI, INIT, BINIT, MCERR, A20M#
- Faults, Aborts (including Machine Check)
- Architectural TLB invalidations including writes to CR0, CR3, CR4, and certain MSR writes; execution of LMSW (occurring prior to issuing MWAIT but after setting the monitor)
- Voluntary transitions due to fast system call and far calls (occurring prior to issuing MWAIT but after setting the monitor)

Power management related events (such as Thermal Monitor 2 or chipset driven STPCLK# assertion) will not cause the monitor event pending flag to be cleared. Faults will not cause the monitor event pending flag to be cleared.

Software should not allow for voluntary context switches in between MONITOR/MWAIT in the instruction flow. Note that execution of MWAIT does not re-arm the monitor hardware. This means that MONITOR/MWAIT need to be executed in a loop. Also note that exits from the MWAIT state could be due to a condition other than a write to the triggering address; software should explicitly check the triggering data location to determine if the write occurred. Software should also check the value of the triggering address following the execution of the monitor instruction (and prior to the execution of the MWAIT instruction). This check is to identify any writes to the triggering address that occurred during the course of MONITOR execution.

The address range provided to the MONITOR instruction must be of write-back caching type. Only write-back memory type stores to the monitored address range will trigger the monitor hardware. If the address range is not in memory of write-back type, the address monitor hardware may not be set up properly or the monitor hardware may not be armed. Software is also responsible for ensuring that

- Writes that are not intended to cause the exit of a busy loop do not write to a location within the address region being monitored by the monitor hardware,
- Writes intended to cause the exit of a busy loop are written to locations within the monitored address region.

Not doing so will lead to more false wakeups (an exit from the MWAIT state not due to a write to the intended data location). These have negative performance implications. It might be necessary for software to use padding to prevent false wakeups. CPUID provides a mechanism for determining the size data locations for monitoring as well as a mechanism for determining the size of a the pad.

10.10.5 Monitor/Mwait Address Range Determination

To use the MONITOR/MWAIT instructions, software should know the length of the region monitored by the MONITOR/MWAIT instructions and the size of the coherence line size for cache-snoop traffic in a multiprocessor system. This information can be queried using the CPUID monitor leaf function (EAX = 05H). You will need the smallest and largest monitor line size:

- To avoid missed wake-ups: make sure that the data structure used to monitor writes fits within the smallest monitor line-size. Otherwise, the processor may not wake up after a write intended to trigger an exit from MWAIT.
- To avoid false wake-ups; use the largest monitor line size to pad the data structure used to monitor writes. Software must make sure that beyond the data structure, no unrelated data variable exists in the triggering area for MWAIT. A pad may be needed to avoid this situation.

These above two values bear no relationship to cache line size in the system and software should not make any assumptions to that effect. Within a single-cluster system, the two parameters should default to be the same (the size of the monitor triggering area is the same as the system coherence line size).

Based on the monitor line sizes returned by the CPUID, the OS should dynamically allocate structures with appropriate padding. If static data structures must be used by an OS, attempt to adapt the data structure and use a

dynamically allocated data buffer for thread synchronization. When the latter technique is not possible, consider not using MONITOR/MWAIT when using static data structures.

To set up the data structure correctly for MONITOR/MWAIT on multi-clustered systems: interaction between processors, chipsets, and the BIOS is required (system coherence line size may depend on the chipset used in the system; the size could be different from the processor's monitor triggering area). The BIOS is responsible to set the correct value for system coherence line size using the IA32_MONITOR_FILTER_LINE_SIZE MSR. Depending on the relative magnitude of the size of the monitor triggering area versus the value written into the IA32_MONITOR_FILTER_LINE_SIZE MSR, the smaller of the parameters will be reported as the *Smallest Monitor Line Size*. The larger of the parameters will be reported as the *Largest Monitor Line Size*.

10.10.6 Required Operating System Support

This section describes changes that must be made to an operating system to run on processors supporting Intel Hyper-Threading Technology. It also describes optimizations that can help an operating system make more efficient use of the logical processors sharing execution resources. The required changes and suggested optimizations are representative of the types of modifications that appear in Windows* XP and Linux* kernel 2.4.0 operating systems for Intel processors supporting Intel Hyper-Threading Technology. Additional optimizations for processors supporting Intel Hyper-Threading Technology are described in the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

10.10.6.1 Use the PAUSE Instruction in Spin-Wait Loops

Intel recommends that a PAUSE instruction be placed in all spin-wait loops that run on Intel processors supporting Intel Hyper-Threading Technology and multi-core processors.

Software routines that use spin-wait loops include multiprocessor synchronization primitives (spin-locks, semaphores, and mutex variables) and idle loops. Such routines keep the processor core busy executing a load-compare-branch loop while a thread waits for a resource to become available. Including a PAUSE instruction in such a loop greatly improves efficiency (see Section 10.10.2, "PAUSE Instruction"). The following routine gives an example of a spin-wait loop that uses a PAUSE instruction:

```
Spin_Lock:
        CMP lockvar, 0      ;Check if lock is free
        JE Get_Lock
        PAUSE               ;Short delay
        JMP Spin_Lock
Get_Lock:
        MOV EAX, 1
        XCHG EAX, lockvar ;Try to get lock
        CMP EAX, 0          ;Test if successful
        JNE Spin_Lock
Critical_Section:
        <critical section code>
        MOV lockvar, 0
        ...
Continue:
```

The spin-wait loop above uses a "test, test-and-set" technique for determining the availability of the synchronization variable. This technique is recommended when writing spin-wait loops.

In IA-32 processor generations earlier than the Pentium 4 processor, the PAUSE instruction is treated as a NOP instruction.

10.10.6.2 Potential Usage of MONITOR/MWAIT in C0 Idle Loops

An operating system may implement different handlers for different idle states. A typical OS idle loop on an ACPI-compatible OS is shown in Example 10-25:

Example 10-25. A Typical OS Idle Loop

```
// WorkQueue is a memory location indicating there is a thread
// ready to run.  A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The idle loop is entered with interrupts disabled.

WHILE (1) {
    IF (WorkQueue) THEN {
        // Schedule work at WorkQueue.
        }
    ELSE {
        // No work to do - wait in appropriate C-state handler depending
        // on Idle time accumulated
        IF (IdleTime >= IdleTimeThreshhold) THEN {
            // Call appropriate C1, C2, C3 state handler, C1 handler
            // shown below
            }
        }
}
// C1 handler uses a Halt instruction
VOID C1Handler()
{   STI
    HLT
}
```

The MONITOR and MWAIT instructions may be considered for use in the C0 idle state loops, if MONITOR and MWAIT are supported.

Example 10-26. An OS Idle Loop with MONITOR/MWAIT in the C0 Idle Loop

```
// WorkQueue is a memory location indicating there is a thread
// ready to run.  A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The following example assumes that the necessary padding has been
// added surrounding WorkQueue to eliminate false wakeups
// The idle loop is entered with interrupts disabled.

WHILE (1) {
    IF (WorkQueue) THEN {
        // Schedule work at WorkQueue.
        }
    ELSE {
        // No work to do - wait in appropriate C-state handler depending
        // on Idle time accumulated.
        IF (IdleTime >= IdleTimeThreshhold) THEN {
            // Call appropriate C1, C2, C3 state handler, C1
            // handler shown below
            MONITOR WorkQueue   // Setup of eax with WorkQueue
                                // LinearAddress,
                                // ECX, EDX = 0
            IF (WorkQueue = 0) THEN {
                MWAIT
                }
            }
        }
```

```
}
// C1 handler uses a Halt instruction.
VOID C1Handler()
{   STI
    HLT
}
```

10.10.6.3 Halt Idle Logical Processors

If one of two logical processors is idle or in a spin-wait loop of long duration, explicitly halt that processor by means of a HLT instruction.

In an MP system, operating systems can place idle processors into a loop that continuously checks the run queue for runnable software tasks. Logical processors that execute idle loops consume a significant amount of core's execution resources that might otherwise be used by the other logical processors in the physical package. For this reason, halting idle logical processors optimizes the performance.[1] If all logical processors within a physical package are halted, the processor will enter a power-saving state.

10.10.6.4 Potential Usage of MONITOR/MWAIT in C1 Idle Loops

An operating system may also consider replacing HLT with MONITOR/MWAIT in its C1 idle loop. An example is shown in Example 10-27:

Example 10-27. An OS Idle Loop with MONITOR/MWAIT in the C1 Idle Loop

```
// WorkQueue is a memory location indicating there is a thread
// ready to run.  A non-zero value for WorkQueue is assumed to
// indicate the presence of work to be scheduled on the processor.
// The following example assumes that the necessary padding has been
// added surrounding WorkQueue to eliminate false wakeups
// The idle loop is entered with interrupts disabled.

WHILE (1) {
    IF (WorkQueue) THEN {
        // Schedule work at WorkQueue
        }
    ELSE {
        // No work to do - wait in appropriate C-state handler depending
        // on Idle time accumulated
        IF (IdleTime >= IdleTimeThreshhold) THEN {
        // Call appropriate C1, C2, C3 state handler, C1
        // handler shown below
        }
    }
}

VOID C1Handler()

{   MONITOR WorkQueue   // Setup of eax with WorkQueue LinearAddress,
                        // ECX, EDX = 0
    IF (WorkQueue = 0) THEN {
        STI
```

1. Excessive transitions into and out of the HALT state could also incur performance penalties. Operating systems should evaluate the performance trade-offs for their operating system.

```
        MWAIT       // EAX, ECX = 0
        }
}
```

10.10.6.5 Guidelines for Scheduling Threads on Logical Processors Sharing Execution Resources

Because the logical processors, the order in which threads are dispatched to logical processors for execution can affect the overall efficiency of a system. The following guidelines are recommended for scheduling threads for execution.

- Dispatch threads to one logical processor per processor core before dispatching threads to the other logical processor sharing execution resources in the same processor core.
- In an MP system with two or more physical packages, distribute threads out over all the physical processors, rather than concentrate them in one or two physical processors.
- Use processor affinity to assign a thread to a specific processor core or package, depending on the cache-sharing topology. The practice increases the chance that the processor's caches will contain some of the thread's code and data when it is dispatched for execution after being suspended.

10.10.6.6 Eliminate Execution-Based Timing Loops

Intel discourages the use of timing loops that depend on a processor's execution speed to measure time. There are several reasons:

- Timing loops cause problems when they are calibrated on a IA-32 processor running at one frequency and then executed on a processor running at another frequency.
- Routines for calibrating execution-based timing loops produce unpredictable results when run on an IA-32 processor supporting Intel Hyper-Threading Technology. This is due to the sharing of execution resources between the logical processors within a physical package.

To avoid the problems described, timing loop routines must use a timing mechanism for the loop that does not depend on the execution speed of the logical processors in the system. The following sources are generally available:

- A high resolution system timer (for example, an Intel 8254).
- A high resolution timer within the processor (such as, the local APIC timer or the time-stamp counter).

For additional information, see the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

10.10.6.7 Place Locks and Semaphores in Aligned, 128-Byte Blocks of Memory

When software uses locks or semaphores to synchronize processes, threads, or other code sections; Intel recommends that only one lock or semaphore be present within a cache line (or 128 byte sector, if 128-byte sector is supported). In processors based on Intel NetBurst microarchitecture (which support 128-byte sector consisting of two cache lines), following this recommendation means that each lock or semaphore should be contained in a 128-byte block of memory that begins on a 128-byte boundary. The practice minimizes the bus traffic required to service locks.

10.11 MP INITIALIZATION FOR P6 FAMILY PROCESSORS

This section describes the MP initialization process for systems that use multiple P6 family processors. This process uses the MP initialization protocol that was introduced with the Pentium Pro processor (see Section 10.4, "Multiple-Processor (MP) Initialization"). For P6 family processors, this protocol is typically used to boot 2 or 4 processors that reside on single system bus; however, it can support from 2 to 15 processors in a multi-clustered system when the APIC buses are tied together. Larger systems are not supported.

10.11.1 Overview of the MP Initialization Process for P6 Family Processors

During the execution of the MP initialization protocol, one processor is selected as the bootstrap processor (BSP) and the remaining processors are designated as application processors (APs), see Section 10.4.1, "BSP and AP Processors." Thereafter, the BSP manages the initialization of itself and the APs. This initialization includes executing BIOS initialization code and operating-system initialization code.

The MP protocol imposes the following requirements and restrictions on the system:

- An APIC clock (APICLK) must be provided.
- The MP protocol will be executed only after a power-up or RESET. If the MP protocol has been completed and a BSP has been chosen, subsequent INITs (either to a specific processor or system wide) do not cause the MP protocol to be repeated. Instead, each processor examines its BSP flag (in the APIC_BASE MSR) to determine whether it should execute the BIOS boot-strap code (if it is the BSP) or enter a wait-for-SIPI state (if it is an AP).
- All devices in the system that are capable of delivering interrupts to the processors must be inhibited from doing so for the duration of the MP initialization protocol. The time during which interrupts must be inhibited includes the window between when the BSP issues an INIT-SIPI-SIPI sequence to an AP and when the AP responds to the last SIPI in the sequence.

The following special-purpose interprocessor interrupts (IPIs) are used during the boot phase of the MP initialization protocol. These IPIs are broadcast on the APIC bus.

- Boot IPI (BIPI)—Initiates the arbitration mechanism that selects a BSP from the group of processors on the system bus and designates the remainder of the processors as APs. Each processor on the system bus broadcasts a BIPI to all the processors following a power-up or RESET.
- Final Boot IPI (FIPI)—Initiates the BIOS initialization procedure for the BSP. This IPI is broadcast to all the processors on the system bus, but only the BSP responds to it. The BSP responds by beginning execution of the BIOS initialization code at the reset vector.
- Startup IPI (SIPI)—Initiates the initialization procedure for an AP. The SIPI message contains a vector to the AP initialization code in the BIOS.

Table 10-5 describes the various fields of the boot phase IPIs.

Table 10-5. Boot Phase IPI Message Format

Type	Destination Field	Destination Shorthand	Trigger Mode	Level	Destination Mode	Delivery Mode	Vector (Hex)
BIPI	Not used	All including self	Edge	Deassert	Don't Care	Fixed (000)	40 to 4E*
FIPI	Not used	All including self	Edge	Deassert	Don't Care	Fixed (000)	10
SIPI	Used	All excluding self	Edge	Assert	Physical	StartUp (110)	00 to FF

NOTE:

* For all P6 family processors.

For BIPI messages, the lower 4 bits of the vector field contain the APIC ID of the processor issuing the message and the upper 4 bits contain the "generation ID" of the message. All P6 family processor will have a generation ID of 4H. BIPIs will therefore use vector values ranging from 40H to 4EH (4FH can not be used because FH is not a valid APIC ID).

10.11.2 MP Initialization Protocol Algorithm

Following a power-up or RESET of a system, the P6 family processors in the system execute the MP initialization protocol algorithm to initialize each of the processors on the system bus. In the course of executing this algorithm, the following boot-up and initialization operations are carried out:

1. Each processor on the system bus is assigned a unique APIC ID, based on system topology (see Section 10.4.5, "Identifying Logical Processors in an MP System"). This ID is written into the local APIC ID register for each processor.
2. Each processor executes its internal BIST simultaneously with the other processors on the system bus. Upon completion of the BIST (at T0), each processor broadcasts a BIPI to "all including self" (see Figure 10-8).
3. APIC arbitration hardware causes all the APICs to respond to the BIPIs one at a time (at T1, T2, T3, and T4).
4. When the first BIPI is received (at time T1), each APIC compares the four least significant bits of the BIPI's vector field with its APIC ID. If the vector and APIC ID match, the processor selects itself as the BSP by setting the BSP flag in its IA32_APIC_BASE MSR. If the vector and APIC ID do not match, the processor selects itself as an AP by entering the "wait for SIPI" state. (Note that in Figure 10-8, the BIPI from processor 1 is the first BIPI to be handled, so processor 1 becomes the BSP.)
5. The newly established BSP broadcasts an FIPI message to "all including self." The FIPI is guaranteed to be handled only after the completion of the BIPIs that were issued by the non-BSP processors.

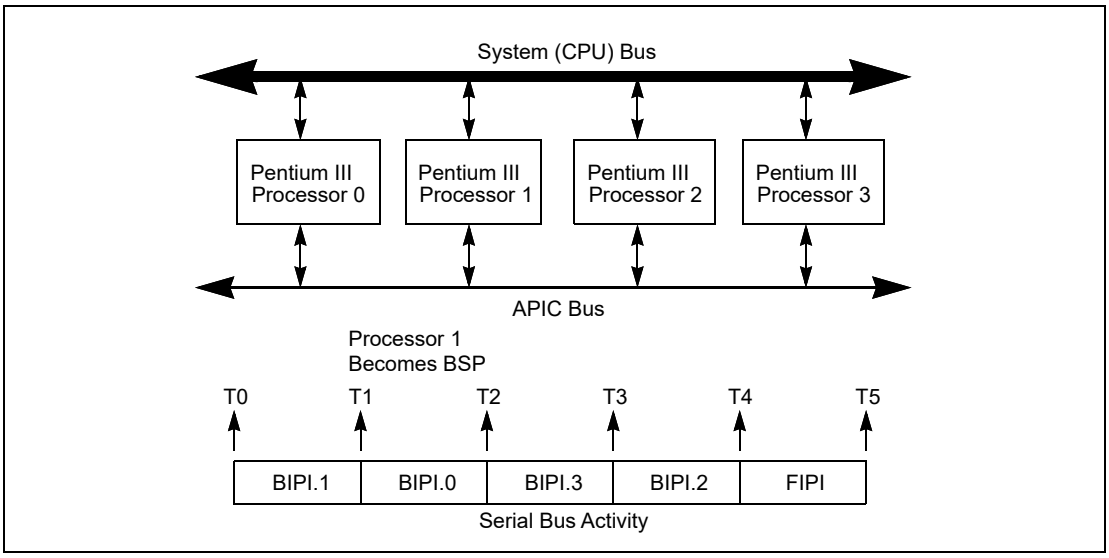


Figure 10-8. MP System With Multiple Pentium III Processors

6. After the BSP has been established, the outstanding BIPIs are received one at a time (at T2, T3, and T4) and ignored by all processors.
7. When the FIPI is finally received (at T5), only the BSP responds to it. It responds by fetching and executing BIOS boot-strap code, beginning at the reset vector (physical address FFFF FFF0H).
8. As part of the boot-strap code, the BSP creates an ACPI table and an MP table and adds its initial APIC ID to these tables as appropriate.
9. At the end of the boot-strap procedure, the BSP broadcasts a SIPI message to all the APs in the system. Here, the SIPI message contains a vector to the BIOS AP initialization code (at 000V V000H, where VV is the vector contained in the SIPI message).
10. All APs respond to the SIPI message by racing to a BIOS initialization semaphore. The first one to the semaphore begins executing the initialization code. (See MP init code for semaphore implementation details.) As part of the AP initialization procedure, the AP adds its APIC ID number to the ACPI and MP tables as appropriate. At the completion of the initialization procedure, the AP executes a CLI instruction (to clear the IF flag in the EFLAGS register) and halts itself.
11. When each of the APs has gained access to the semaphore and executed the AP initialization code and all written their APIC IDs into the appropriate places in the ACPI and MP tables, the BSP establishes a count for the number of processors connected to the system bus, completes executing the BIOS boot-strap code, and then begins executing operating-system boot-strap and start-up code.

12. While the BSP is executing operating-system boot-strap and start-up code, the APs remain in the halted state. In this state they will respond only to INITs, NMIs, and SMIs. They will also respond to snoops and to assertions of the STPCLK# pin.

See Section 10.4.4, "MP Initialization Example," for an annotated example the use of the MP protocol to boot IA-32 processors in an MP. This code should run on any IA-32 processor that used the MP protocol.

10.11.2.1 Error Detection and Handling During the MP Initialization Protocol

Errors may occur on the APIC bus during the MP initialization phase. These errors may be transient or permanent and can be caused by a variety of failure mechanisms (for example, broken traces, soft errors during bus usage, etc.). All serial bus related errors will result in an APIC checksum or acceptance error.

The MP initialization protocol makes the following assumptions regarding errors that occur during initialization:

- If errors are detected on the APIC bus during execution of the MP initialization protocol, the processors that detect the errors are shut down.
- The MP initialization protocol will be executed by processors even if they fail their BIST sequences.

CHAPTER 11
PROCESSOR MANAGEMENT AND INITIALIZATION

This chapter describes the facilities provided for managing processor wide functions and for initializing the processor. The subjects covered include: processor initialization, x87 FPU initialization, processor configuration, feature determination, mode switching, the MSRs (in the Pentium, P6 family, Pentium 4, and Intel Xeon processors), and the MTRRs (in the P6 family, Pentium 4, and Intel Xeon processors).

11.1 INITIALIZATION OVERVIEW

Following power-up or an assertion of the RESET# pin, each processor on the system bus performs a hardware initialization of the processor (known as a hardware reset) and an optional built-in self-test (BIST). A hardware reset sets each processor's registers to a known state and places the processor in real-address mode. It also invalidates the internal caches, translation lookaside buffers (TLBs) and the branch target buffer (BTB). At this point, the action taken depends on the processor family:

- **Pentium 4 processors (CPUID DisplayFamily 0FH)** — All the processors on the system bus (including a single processor in a uniprocessor system) execute the multiple processor (MP) initialization protocol. The processor that is selected through this protocol as the bootstrap processor (BSP) then immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. The application (non-BSP) processors (APs) go into a Wait For Startup IPI (SIPI) state while the BSP is executing initialization code. See Section 10.4, "Multiple-Processor (MP) Initialization," for more details. Note that in a uniprocessor system, the single Pentium 4 or Intel Xeon processor automatically becomes the BSP.
- **IA-32 and Intel 64 processors (CPUID DisplayFamily 06H)** — The action taken is the same as for the Pentium 4 processors (as described in the previous paragraph).
- **Pentium processors** — In either a single- or dual- processor system, a single Pentium processor is always pre-designated as the primary processor. Following a reset, the primary processor behaves as follows in both single- and dual-processor systems. Using the dual-processor (DP) ready initialization protocol, the primary processor immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. The secondary processor (if there is one) goes into a halt state.
- **Intel486 processor** — The primary processor (or single processor in a uniprocessor system) immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. (The Intel486 does not automatically execute a DP or MP initialization protocol to determine which processor is the primary processor.)

The software-initialization code performs all system-specific initialization of the BSP or primary processor and the system logic.

At this point, for MP (or DP) systems, the BSP (or primary) processor wakes up each AP (or secondary) processor to enable those processors to execute self-configuration code.

When all processors are initialized, configured, and synchronized, the BSP or primary processor begins executing an initial operating-system or executive task.

The x87 FPU is also initialized to a known state during hardware reset. x87 FPU software initialization code can then be executed to perform operations such as setting the precision of the x87 FPU and the exception masks. No special initialization of the x87 FPU is required to switch operating modes.

Asserting the INIT# pin on the processor invokes a similar response to a hardware reset. The major difference is that during an INIT, the internal caches, MSRs, MTRRs, and x87 FPU state are left unchanged (although, the TLBs and BTB are invalidated as with a hardware reset). An INIT provides a method for switching from protected to real-address mode while maintaining the contents of the internal caches.

11.1.1 Processor State After Reset

Following power-up, The state of control register CR0 is 60000010H (see Figure 11-1). This places the processor is in real-address mode with paging disabled.

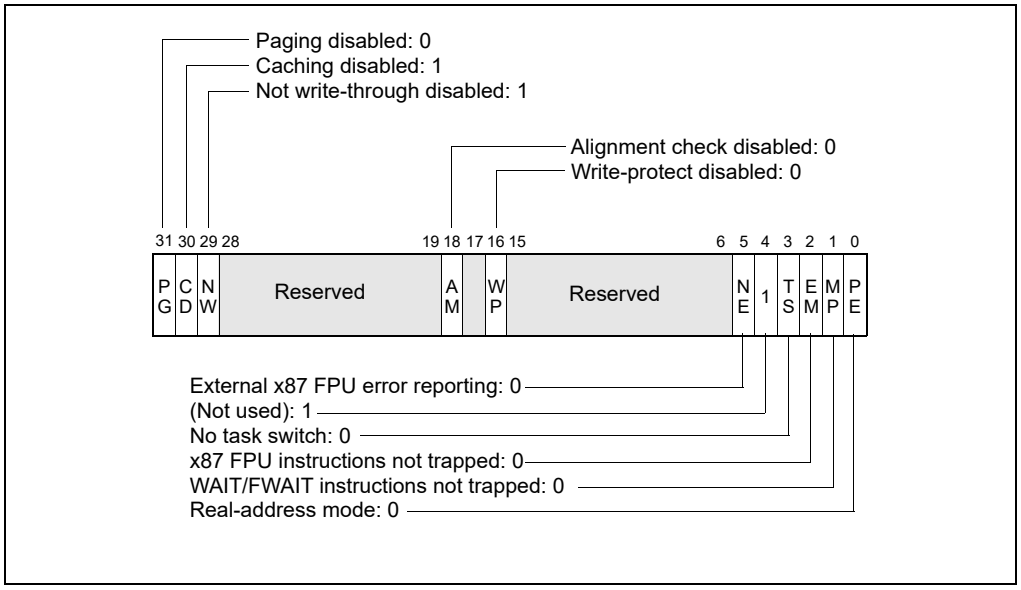


Figure 11-1. Contents of CR0 Register after Reset

The state of the flags and other registers following power-up for the Pentium 4, Pentium Pro, and Pentium processors are shown in Section 24.39, "Initial State of Pentium, Pentium Pro and Pentium 4 Processors," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

Table 11-1 shows processor states of IA-32 and Intel 64 processors with CPUID DisplayFamily signature of 06H at the following events: power-up, RESET, and INIT. In a few cases, the behavior of some registers behave slightly different across warm RESET, the variant cases are marked in Table 11-1 and described in more detail in Table 11-2.

Table 11-1. IA-32 and Intel® 64 Processor States Following Power-up, Reset, or INIT

Register	Power up	Reset	INIT
EFLAGS[1]	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H[2]	60000010H[2]	60000010H[2]
CR2, CR3, CR4	00000000H	00000000H	00000000H
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	000n06xxH[3]	000n06xxH[3]	000n06xxH[3]
EAX	0[4]	0[4]	0[4]
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H

Table 11-1. IA-32 and Intel® 64 Processor States Following Power-up, Reset, or INIT (Contd.)

Register	Power up	Reset	INIT
ST0 through ST7[5]	+0.0	+0.0	FINIT/FNINIT: Unchanged
x87 FPU Control Word[5]	0040H	0040H	FINIT/FNINIT: 037FH
x87 FPU Status Word[5]	0000H	0000H	FINIT/FNINIT: 0000H
x87 FPU Tag Word[5]	5555H	5555H	FINIT/FNINIT: FFFFH
x87 FPU Data Operand and CS Seg. Selectors[5]	0000H	0000H	FINIT/FNINIT: 0000H
x87 FPU Data Operand and Inst. Pointers[5]	00000000H	00000000H	FINIT/FNINIT: 00000000H
MM0 through MM7[5]	0000000000000000H	0000000000000000H	INIT or FINIT/FNINIT: Unchanged
XMM0 through XMM7	0H	0H	Unchanged
MXCSR	1F80H	1F80H	Unchanged
GDTR, IDTR	Base = 00000000H Limit = FFFFH AR = Present, R/W	Base = 00000000H Limit = FFFFH AR = Present, R/W	Base = 00000000H Limit = FFFFH AR = Present, R/W
LDTR, Task Register	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W
DR0, DR1, DR2, DR3	00000000H	00000000H	00000000H
DR6	FFFF0FF0H	FFFF0FF0H	FFFF0FF0H
DR7	00000400H	00000400H	00000400H
R8-R15	0000000000000000H	0000000000000000H	0000000000000000H
XMM8-XMM15	0H	0H	Unchanged
XCR0	1H	1H	Unchanged
IA32_XSS	0H	0H	Unchanged
YMM_H[255:128]	0H	0H	Unchanged
BNDCFGU	0H	0H	0H
BND0-BND3	0H	0H	0H
IA32_BNDCFGS	0H	0H	0H
OPMASK	0H	0H	Unchanged
ZMM_H[511:256]	0H	0H	Unchanged
ZMMHi16[511:0]	0H	0H	Unchanged
PKRU	0H	0H	Unchanged
Intel Processor Trace MSRs	0H	0H[W]	Unchanged
Time-Stamp Counter	0H	0H[W]	Unchanged
IA32_TSC_AUX	0H	0H	Unchanged
IA32_TSC_ADJUST	0H	0H	Unchanged
IA32_TSC_DEADLINE	0H	0H	Unchanged
IA32_SYSENTER_CS/ESP/EIP	0H	0H	Unchanged
IA32_EFER	0000000000000000H	0000000000000000H	0000000000000000H
IA32_STAR/LSTAR	0H	0H	Unchanged

Table 11-1. IA-32 and Intel® 64 Processor States Following Power-up, Reset, or INIT (Contd.)

Register	Power up	Reset	INIT
IA32_FS_BASE/GS_BASE	0H	0H	0H
IA32_PMCx, IA32_PERFEVTSELx	0H	0H	Unchanged
IA32_PERF_GLOBAL_CTRL	Sets bits n-1:0 and clears the upper bits.[7]	Sets bits n-1:0 and clears the upper bits.[7]	Unchanged
IA32_FIXED_CTRx, IA32_FIXED_CTR_CTRL	0H	0H	Unchanged
Data and Code Cache, TLBs	Invalid[6]	Invalid[6]	Unchanged
Fixed MTRRs	Disabled	Disabled	Unchanged
Variable MTRRs	Disabled	Disabled	Unchanged
Machine-Check Banks	Undefined	Undefined[W]	Unchanged
Last Branch Record Stack	0	0[W]	Unchanged
APIC	Enabled	Enabled	Unchanged
X2APIC	Disabled	Disabled	Unchanged
IA32_DEBUG_INTERFACE	0	0[W]	Unchanged

NOTES:

1. The 10 most-significant bits of the EFLAGS register are undefined following a reset. Software should not depend on the states of any of these bits.
2. The CD and NW flags are unchanged, bit 4 is set to 1, all other bits are cleared.
3. Where "n" is the Extended Model Value for the respective processor, and "xx" = don't care.
4. If Built-In Self-Test (BIST) is invoked on power up or reset, EAX is 0 only if all tests passed. (BIST cannot be invoked during an INIT.)
5. The state of the x87 FPU and MMX registers is not changed by the execution of an INIT.
6. Internal caches are invalid after power-up and RESET, but left unchanged with an INIT.
7. Where "n" is the number of general-purpose counters available in the processor. See Chapter 21, "Performance Monitoring," for additional details.

W: Warm RESET behavior differs from power-on RESET with details listed in Table 11-2.

Table 11-2. Variance of RESET Values in Selected Intel Architecture Processors

State	XREF	Value	Feature Flag or DisplayFamily_DisplayModel Signatures
Time-Stamp Counter	Warm RESET	Unmodified across warm Reset	06_2DH, 06_3EH
Machine-Check Banks	Warm RESET	IA32_MCi_Status banks are unmodified across warm Reset	06_2DH, 06_3EH, 06_3FH, 06_4FH, 06_56H
Last Branch Record Stack	Warm RESET	LBR stack MSRs are unmodified across warm Reset	06_1AH, 06_1CH, DisplayFamiy= 06 and DisplayModel >1DH
Intel Processor Trace MSRs	Warm RESET	Clears IA32_RTIT_CTL.TraceEn, the rest of MSRs are unmodified	If CPUID.(EAX=14H, ECX=0H):EBX[bit 2] = 1
IA32_DEBUG_INTERFACE	Warm RESET	Unmodified across warm Reset	If CPUID.01H:ECX.[11] = 1

11.1.2 Processor Built-In Self-Test (BIST)

Hardware may request that the BIST be performed at power-up. The EAX register is cleared (0H) if the processor passes the BIST. A nonzero value in the EAX register after the BIST indicates that a processor fault was detected. If the BIST is not requested, the contents of the EAX register after a hardware reset is 0H.

The overhead for performing a BIST varies between processor families. For example, the BIST takes approximately 30 million processor clock periods to execute on the Pentium 4 processor. This clock count is model-specific; Intel reserves the right to change the number of periods for any Intel 64 or IA-32 processor, without notification.

11.1.3 Model and Stepping Information

Following a hardware reset, the EDX register contains component identification and revision information (see Figure 11-2). For example, the model, family, and processor type returned for the first processor in the Intel Pentium 4 family is as follows: model (0000B), family (1111B), and processor type (00B).

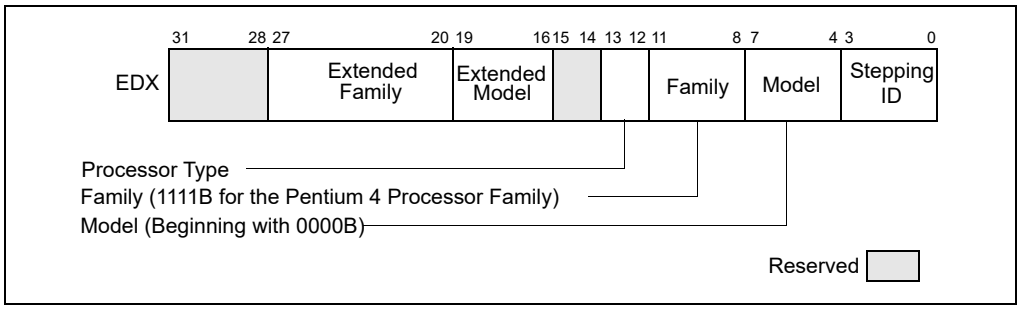


Figure 11-2. Version Information in the EDX Register after Reset

The stepping ID field contains a unique identifier for the processor's stepping ID or revision level. The extended family and extended model fields were added to the IA-32 architecture in the Pentium 4 processors.

11.1.4 First Instruction Executed

The first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFF0H. This address is 16 bytes below the processor's uppermost physical address. The EPROM containing the software-initialization code must be located at this address.

The address FFFFFFF0H is beyond the 1-MByte addressable range of the processor while in real-address mode. The processor is initialized to this starting address as follows. The CS register has two parts: the visible segment selector part and the hidden base address part. In real-address mode, the base address is normally formed by shifting the 16-bit segment selector value 4 bits to the left to produce a 20-bit base address. However, during a hardware reset, the segment selector in the CS register is loaded with F000H and the base address is loaded with FFFF0000H. The starting address is thus formed by adding the base address to the value in the EIP register (that is, FFFF0000 + FFF0H = FFFFFFF0H).

The first time the CS register is loaded with a new value after a hardware reset, the processor will follow the normal rule for address translation in real-address mode (that is, [CS base address = CS segment selector * 16]). To ensure that the base address in the CS register remains unchanged until the EPROM based software-initialization code is completed, the code must not contain a far jump or far call or allow an interrupt to occur (which would cause the CS selector value to be changed).

11.2 X87 FPU INITIALIZATION

Software-initialization code can determine the whether the processor contains an x87 FPU by using the CPUID instruction. The code must then initialize the x87 FPU and set flags in control register CR0 to reflect the state of the x87 FPU environment.

A hardware reset places the x87 FPU in the state shown in Table 11-1. This state is different from the state the x87 FPU is placed in following the execution of an FINIT or FNINIT instruction (also shown in Table 11-1). If the x87 FPU is to be used, the software-initialization code should execute an FINIT/FNINIT instruction following a hardware reset. These instructions, tag all data registers as empty, clear all the exception masks, set the TOP-of-stack value to 0, and select the default rounding and precision controls setting (round to nearest and 64-bit precision).

If the processor is reset by asserting the INIT# pin, the x87 FPU state is not changed.

11.2.1 Configuring the x87 FPU Environment

Initialization code must load the appropriate values into the MP, EM, and NE flags of control register CR0. These bits are cleared on hardware reset of the processor. Figure 11-3 shows the suggested settings for these flags, depending on the IA-32 processor being initialized. Initialization code can test for the type of processor present before setting or clearing these flags.

Table 11-3. Recommended Settings of EM and MP Flags on IA-32 Processors

EM	MP	NE	IA-32 processor
1	0	1	Intel486™ SX, Intel386™ DX, and Intel386™ SX processors only, without the presence of a math coprocessor.
0	1	1 or 0*	Pentium 4, Intel Xeon, P6 family, Pentium, Intel486™ DX, and Intel 487 SX processors, and Intel386 DX and Intel386 SX processors when a companion math coprocessor is present.
0	1	1 or 0*	More recent Intel 64 or IA-32 processors

NOTE:

* The setting of the NE flag depends on the operating system being used.

The EM flag determines whether floating-point instructions are executed by the x87 FPU (EM is cleared) or a device-not-available exception (#NM) is generated for all floating-point instructions so that an exception handler can emulate the floating-point operation (EM = 1). Ordinarily, the EM flag is cleared when an x87 FPU or math coprocessor is present and set if they are not present. If the EM flag is set and no x87 FPU, math coprocessor, or floating-point emulator is present, the processor will hang when a floating-point instruction is executed.

The MP flag determines whether WAIT/FWAIT instructions react to the setting of the TS flag. If the MP flag is clear, WAIT/FWAIT instructions ignore the setting of the TS flag; if the MP flag is set, they will generate a device-not-available exception (#NM) if the TS flag is set. Generally, the MP flag should be set for processors with an integrated x87 FPU and clear for processors without an integrated x87 FPU and without a math coprocessor present. However, an operating system can choose to save the floating-point context at every context switch, in which case there would be no need to set the MP bit.

Table 2-2 shows the actions taken for floating-point and WAIT/FWAIT instructions based on the settings of the EM, MP, and TS flags.

The NE flag determines whether unmasked floating-point exceptions are handled by generating a floating-point error exception internally (NE is set, native mode) or through an external interrupt (NE is cleared). In systems where an external interrupt controller is used to invoke numeric exception handlers (such as MS-DOS-based systems), the NE bit should be cleared.

11.2.2 Setting the Processor for x87 FPU Software Emulation

Setting the EM flag causes the processor to generate a device-not-available exception (#NM) and trap to a software exception handler whenever it encounters a floating-point instruction. (Table 11-3 shows when it is appropriate to use this flag.) Setting this flag has two functions:

- It allows x87 FPU code to run on an IA-32 processor that has neither an integrated x87 FPU nor is connected to an external math coprocessor, by using a floating-point emulator.
- It allows floating-point code to be executed using a special or nonstandard floating-point emulator, selected for a particular application, regardless of whether an x87 FPU or math coprocessor is present.

To emulate floating-point instructions, the EM, MP, and NE flag in control register CR0 should be set as shown in Table 11-4.

Table 11-4. Software Emulation Settings of EM, MP, and NE Flags

CR0 Bit	Value
EM	1
MP	0
NE	1

Regardless of the value of the EM bit, the Intel486 SX processor generates a device-not-available exception (#NM) upon encountering any floating-point instruction.

11.3 CACHE ENABLING

IA-32 processors (beginning with the Intel486 processor) and Intel 64 processors contain internal instruction and data caches. These caches are enabled by clearing the CD and NW flags in control register CR0. (They are set during a hardware reset.) Because all internal cache lines are invalid following reset initialization, it is not necessary to invalidate the cache before enabling caching. Any external caches may require initialization and invalidation using a system-specific initialization and invalidation code sequence.

Depending on the hardware and operating system or executive requirements, additional configuration of the processor's caching facilities will probably be required. Beginning with the Intel486 processor, page-level caching can be controlled with the PCD and PWT flags in page-directory and page-table entries. Beginning with the P6 family processors, the memory type range registers (MTRRs) control the caching characteristics of the regions of physical memory. (For the Intel486 and Pentium processors, external hardware can be used to control the caching characteristics of regions of physical memory.) See Chapter 13, "Memory Cache Control," for detailed information on configuration of the caching facilities in the Pentium 4, Intel Xeon, and P6 family processors and system memory.

11.4 MODEL-SPECIFIC REGISTERS (MSRS)

Most IA-32 processors (starting from Pentium processors) and Intel 64 processors contain a model-specific registers (MSRs). A given MSR may not be supported across all families and models for Intel 64 and IA-32 processors. Some MSRs are designated as architectural to simplify software programming; a feature introduced by an architectural MSR is expected to be supported in future processors. Non-architectural MSRs are not guaranteed to be supported or to have the same functions on future processors.

MSRs that provide control for a number of hardware and software-related features, include:

- Performance-monitoring counters (see Chapter 21, "Performance Monitoring").
- Debug extensions (see Chapter 19, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features").
- Machine-check exception capability and its accompanying machine-check architecture (see Chapter 17, "Machine-Check Architecture").
- MTRRs (see Section 13.11, "Memory Type Range Registers (MTRRs)").
- Thermal and power management.
- Instruction-specific support (for example: SYSENTER, SYSEXIT, SWAPGS, etc.).
- Processor feature/mode support (for example: IA32_EFER, IA32_FEATURE_CONTROL).

The MSRs can be read and written to using the RDMSR and WRMSR instructions, respectively.

When performing software initialization of an IA-32 or Intel 64 processor, many of the MSRs will need to be initialized to set up things like performance-monitoring events, run-time machine checks, and memory types for physical memory.

Lists of available performance-monitoring events can be found at: https://perfmon-events.intel.com/, and lists of available MSRs are given in Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4. The references earlier in this section show where the functions of the various groups of MSRs are described in this manual.

11.5 MEMORY TYPE RANGE REGISTERS (MTRRS)

Memory type range registers (MTRRs) were introduced into the IA-32 architecture with the Pentium Pro processor. They allow the type of caching (or no caching) to be specified in system memory for selected physical address ranges. They allow memory accesses to be optimized for various types of memory such as RAM, ROM, frame buffer memory, and memory-mapped I/O devices.

In general, initializing the MTRRs is normally handled by the software initialization code or BIOS and is not an operating system or executive function. At the very least, all the MTRRs must be cleared to 0, which selects the uncached (UC) memory type. See Section 13.11, "Memory Type Range Registers (MTRRs)," for detailed information on the MTRRs.

11.6 INITIALIZING SSE/SSE2/SSE3/SSSE3 EXTENSIONS

For processors that contain SSE/SSE2/SSE3/SSSE3 extensions, steps must be taken when initializing the processor to allow execution of these instructions.

1. Check the CPUID feature flags for the presence of the SSE/SSE2/SSE3/SSSE3 extensions (respectively: EDX bits 25 and 26, ECX bit 0 and 9) and support for the FXSAVE and FXRSTOR instructions (EDX bit 24). Also check for support for the CLFLUSH instruction (EDX bit 19). The CPUID feature flags are loaded in the EDX and ECX registers when the CPUID instruction is executed with a 1 in the EAX register.
2. Set the OSFXSR flag (bit 9 in control register CR4) to indicate that the operating system supports saving and restoring the SSE/SSE2/SSE3/SSSE3 execution environment (XMM and MXCSR registers) with the FXSAVE and FXRSTOR instructions, respectively. See Section 2.5, "Control Registers," for a description of the OSFXSR flag.
3. Set the OSXMMEXCPT flag (bit 10 in control register CR4) to indicate that the operating system supports the handling of SSE/SSE2/SSE3 SIMD floating-point exceptions (#XM). See Section 2.5, "Control Registers," for a description of the OSXMMEXCPT flag.
4. Set the mask bits and flags in the MXCSR register according to the mode of operation desired for SSE/SSE2/SSE3 SIMD floating-point instructions. See "MXCSR Control and Status Register" in Chapter 10, "Programming with Intel® Streaming SIMD Extensions (Intel® SSE)," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for a detailed description of the bits and flags in the MXCSR register.

11.7 SOFTWARE INITIALIZATION FOR REAL-ADDRESS MODE OPERATION

Following a hardware reset (either through a power-up or the assertion of the RESET# pin) the processor is placed in real-address mode and begins executing software initialization code from physical address FFFFFFF0H. Software initialization code must first set up the necessary data structures for handling basic system functions, such as a real-mode IDT for handling interrupts and exceptions. If the processor is to remain in real-address mode, software must then load additional operating-system or executive code modules and data structures to allow reliable execution of application programs in real-address mode.

If the processor is going to operate in protected mode, software must load the necessary data structures to operate in protected mode and then switch to protected mode. The protected-mode data structures that must be loaded are described in Section 11.8, "Software Initialization for Protected-Mode Operation."

11.7.1 Real-Address Mode IDT

In real-address mode, the only system data structure that must be loaded into memory is the IDT (also called the "interrupt vector table"). By default, the address of the base of the IDT is physical address 0H. This address can be

changed by using the LIDT instruction to change the base address value in the IDTR. Software initialization code needs to load interrupt- and exception-handler pointers into the IDT before interrupts can be enabled.

The actual interrupt- and exception-handler code can be contained either in EPROM or RAM; however, the code must be located within the 1-MByte addressable range of the processor in real-address mode. If the handler code is to be stored in RAM, it must be loaded along with the IDT.

11.7.2 NMI Interrupt Handling

The NMI interrupt is always enabled (except when multiple NMIs are nested). If the IDT and the NMI interrupt handler need to be loaded into RAM, there will be a period of time following hardware reset when an NMI interrupt cannot be handled. During this time, hardware must provide a mechanism to prevent an NMI interrupt from halting code execution until the IDT and the necessary NMI handler software is loaded. Here are two examples of how NMIs can be handled during the initial states of processor initialization:

- A simple IDT and NMI interrupt handler can be provided in EPROM. This allows an NMI interrupt to be handled immediately after reset initialization.
- The system hardware can provide a mechanism to enable and disable NMIs by passing the NMI# signal through an AND gate controlled by a flag in an I/O port. Hardware can clear the flag when the processor is reset, and software can set the flag when it is ready to handle NMI interrupts.

11.8 SOFTWARE INITIALIZATION FOR PROTECTED-MODE OPERATION

The processor is placed in real-address mode following a hardware reset. At this point in the initialization process, some basic data structures and code modules must be loaded into physical memory to support further initialization of the processor, as described in Section 11.7, “Software Initialization for Real-Address Mode Operation.” Before the processor can be switched to protected mode, the software initialization code must load a minimum number of protected mode data structures and code modules into memory to support reliable operation of the processor in protected mode. These data structures include the following:

- A IDT.
- A GDT.
- A TSS.
- (Optional) An LDT.
- If paging is to be used, at least one page directory and one page table.
- A code segment that contains the code to be executed when the processor switches to protected mode.
- One or more code modules that contain the necessary interrupt and exception handlers.

Software initialization code must also initialize the following system registers before the processor can be switched to protected mode:

- The GDTR.
- (Optional.) The IDTR. This register can also be initialized immediately after switching to protected mode, prior to enabling interrupts.
- Control registers CR1 through CR4.
- (Pentium 4, Intel Xeon, and P6 family processors only.) The memory type range registers (MTRRs).

With these data structures, code modules, and system registers initialized, the processor can be switched to protected mode by loading control register CR0 with a value that sets the PE flag (bit 0).

11.8.1 Protected-Mode System Data Structures

The contents of the protected-mode system data structures loaded into memory during software initialization, depend largely on the type of memory management the protected-mode operating-system or executive is going to support: flat, flat with paging, segmented, or segmented with paging.

To implement a flat memory model without paging, software initialization code must at a minimum load a GDT with one code and one data-segment descriptor. A null descriptor in the first GDT entry is also required. The stack can be placed in a normal read/write data segment, so no dedicated descriptor for the stack is required. A flat memory model with paging also requires a page directory and at least one page table (unless all pages are 4 MBytes in which case only a page directory is required). See Section 11.8.3, "Initializing Paging."

Before the GDT can be used, the base address and limit for the GDT must be loaded into the GDTR register using an LGDT instruction.

A multi-segmented model may require additional segments for the operating system, as well as segments and LDTs for each application program. LDTs require segment descriptors in the GDT. Some operating systems allocate new segments and LDTs as they are needed. This provides maximum flexibility for handling a dynamic programming environment. However, many operating systems use a single LDT for all tasks, allocating GDT entries in advance. An embedded system, such as a process controller, might pre-allocate a fixed number of segments and LDTs for a fixed number of application programs. This would be a simple and efficient way to structure the software environment of a real-time system.

11.8.2 Initializing Protected-Mode Exceptions and Interrupts

Software initialization code must at a minimum load a protected-mode IDT with gate descriptor for each exception vector that the processor can generate. If interrupt or trap gates are used, the gate descriptors can all point to the same code segment, which contains the necessary exception handlers. If task gates are used, one TSS and accompanying code, data, and task segments are required for each exception handler called with a task gate.

If hardware allows interrupts to be generated, gate descriptors must be provided in the IDT for one or more interrupt handlers.

Before the IDT can be used, the base address and limit for the IDT must be loaded into the IDTR register using an LIDT instruction. This operation is typically carried out immediately after switching to protected mode.

11.8.3 Initializing Paging

Paging is controlled by the PG flag in control register CR0. When this flag is clear (its state following a hardware reset), the paging mechanism is turned off; when it is set, paging is enabled. Before setting the PG flag, the following data structures and registers must be initialized:

- Software must load at least one page directory and one page table into physical memory. The page table can be eliminated if the page directory contains a directory entry pointing to itself (here, the page directory and page table reside in the same page), or if only 4-MByte pages are used.
- Control register CR3 (also called the PDBR register) is loaded with the physical base address of the page directory.
- (Optional) Software may provide one set of code and data descriptors in the GDT or in an LDT for supervisor mode and another set for user mode.

With this paging initialization complete, paging is enabled and the processor is switched to protected mode at the same time by loading control register CR0 with an image in which the PG and PE flags are set. (Paging cannot be enabled before the processor is switched to protected mode.)

11.8.4 Initializing Multitasking

If the multitasking mechanism is not going to be used and changes between privilege levels are not allowed, it is not necessary load a TSS into memory or to initialize the task register.

If the multitasking mechanism is going to be used and/or changes between privilege levels are allowed, software initialization code must load at least one TSS and an accompanying TSS descriptor. (A TSS is required to change privilege levels because pointers to the privileged-level 0, 1, and 2 stack segments and the stack pointers for these stacks are obtained from the TSS.) TSS descriptors must not be marked as busy when they are created; they should be marked busy by the processor only as a side-effect of performing a task switch. As with descriptors for LDTs, TSS descriptors reside in the GDT.

After the processor has switched to protected mode, the LTR instruction can be used to load a segment selector for a TSS descriptor into the task register. This instruction marks the TSS descriptor as busy, but does not perform a task switch. The processor can, however, use the TSS to locate pointers to privilege-level 0, 1, and 2 stacks. The segment selector for the TSS must be loaded before software performs its first task switch in protected mode, because a task switch copies the current task state into the TSS.

After the LTR instruction has been executed, further operations on the task register are performed by task switching. As with other segments and LDTs, TSSs and TSS descriptors can be either pre-allocated or allocated as needed.

11.8.5 Initializing IA-32e Mode

On Intel 64 processors, the IA32_EFER MSR is cleared on system reset. The operating system must be in protected mode with paging enabled before attempting to initialize IA-32e mode. IA-32e mode operation also requires physical-address extensions with four or five levels of enhanced paging structures (see Section 5.5, "4-Level Paging and 5-Level Paging").

Operating systems should follow this sequence to initialize IA-32e mode:

1. Starting from protected mode, disable paging by setting CR0.PG = 0. Use the MOV CR0 instruction to disable paging (the instruction must be located in an identity-mapped page).
2. Enable physical-address extensions (PAE) by setting CR4.PAE = 1. Failure to enable PAE will result in a #GP fault when an attempt is made to initialize IA-32e mode.
3. Load CR3 with the physical base address of the Level 4 page map table (PML4) or Level 5 page map table (PML5).
4. Enable IA-32e mode by setting IA32_EFER.LME = 1.
5. Enable paging by setting CR0.PG = 1. This causes the processor to set the IA32_EFER.LMA bit to 1. The MOV CR0 instruction that enables paging and the following instructions must be located in an identity-mapped page (until such time that a branch to non-identity mapped pages can be effected).

64-bit mode paging structures must be located in the first 4 GBytes of physical-address space prior to activating IA-32e mode. This is necessary because the MOV CR3 instruction used to initialize the page-directory base must be executed in legacy mode prior to activating IA-32e mode (setting CR0.PG = 1 to enable paging). Because MOV CR3 is executed in protected mode, only the lower 32 bits of the register are written, limiting the table location to the low 4 GBytes of memory. Software can relocate the page tables anywhere in physical memory after IA-32e mode is activated.

The processor performs 64-bit mode consistency checks whenever software attempts to modify any of the enable bits directly involved in activating IA-32e mode (IA32_EFER.LME, CR0.PG, and CR4.PAE). It will generate a general protection fault (#GP) if consistency checks fail. 64-bit mode consistency checks ensure that the processor does not enter an undefined mode or state with unpredictable behavior.

64-bit mode consistency checks fail in the following circumstances:

- An attempt is made to enable or disable IA-32e mode while paging is enabled.
- IA-32e mode is enabled and an attempt is made to enable paging prior to enabling physical-address extensions (PAE).
- IA-32e mode is active and an attempt is made to disable physical-address extensions (PAE).
- If the current CS has the L-bit set on an attempt to activate IA-32e mode.
- If the TR contains a 16-bit TSS on an attempt to activate IA-32e mode.

11.8.5.1 IA-32e Mode System Data Structures

After activating IA-32e mode, the system-descriptor-table registers (GDTR, LDTR, IDTR, TR) continue to reference legacy protected-mode descriptor tables. Tables referenced by the descriptors all reside in the lower 4 GBytes of linear-address space. After activating IA-32e mode, 64-bit operating-systems should use the LGDT, LLDT, LIDT, and LTR instructions to load the system-descriptor-table registers with references to 64-bit descriptor tables.

11.8.5.2 IA-32e Mode Interrupts and Exceptions

Software must not allow exceptions or interrupts to occur between the time IA-32e mode is activated and the update of the interrupt-descriptor-table register (IDTR) that establishes references to a 64-bit interrupt-descriptor table (IDT). This is because the IDT remains in legacy form immediately after IA-32e mode is activated.

If an interrupt or exception occurs prior to updating the IDTR, a legacy 32-bit interrupt gate will be referenced and interpreted as a 64-bit interrupt gate with unpredictable results. External interrupts can be disabled by using the CLI instruction.

Non-maskable interrupts (NMI) must be disabled using external hardware.

11.8.5.3 64-bit Mode and Compatibility Mode Operation

IA-32e mode uses two code segment-descriptor bits (CS.L and CS.D, see Figure 3-8) to control the operating modes after IA-32e mode is initialized. If CS.L = 1 and CS.D = 0, the processor is running in 64-bit mode. With this encoding, the default operand size is 32 bits and default address size is 64 bits. Using instruction prefixes, operand size can be changed to 64 bits or 16 bits; address size can be changed to 32 bits.

When IA-32e mode is active and CS.L = 0, the processor operates in compatibility mode. In this mode, CS.D controls default operand and address sizes exactly as it does in the IA-32 architecture. Setting CS.D = 1 specifies default operand and address size as 32 bits. Clearing CS.D to 0 specifies default operand and address size as 16 bits (the CS.L = 1, CS.D = 1 bit combination is reserved).

Compatibility mode execution is selected on a code-segment basis. This mode allows legacy applications to coexist with 64-bit applications running in 64-bit mode. An operating system running in IA-32e mode can execute existing 16-bit and 32-bit applications by clearing their code-segment descriptor's CS.L bit to 0.

In compatibility mode, the following system-level mechanisms continue to operate using the IA-32e-mode architectural semantics:

- Linear-to-physical address translation uses the 64-bit mode extended page-translation mechanism.
- Interrupts and exceptions are handled using the 64-bit mode mechanisms.
- System calls (calls through call gates and SYSENTER/SYSEXIT) are handled using the IA-32e mode mechanisms.

11.8.5.4 Switching Out of IA-32e Mode Operation

To return from IA-32e mode to paged-protected mode operation operating systems must use the following sequence:

1. Switch to compatibility mode.
2. Deactivate IA-32e mode by clearing CR0.PG = 0. This causes the processor to set IA32_EFER.LMA = 0. The MOV CR0 instruction used to disable paging and subsequent instructions must be located in an identity-mapped page.
3. Load CR3 with the physical base address of the legacy page-table-directory base address.
4. Disable IA-32e mode by setting IA32_EFER.LME = 0.
5. Enable legacy paged-protected mode by setting CR0.PG = 1
6. A branch instruction must follow the MOV CR0 that enables paging. Both the MOV CR0 and the branch instruction must be located in an identity-mapped page.

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

11.9 MODE SWITCHING

To use the processor in protected mode after hardware or software reset, a mode switch must be performed from real-address mode. Once in protected mode, software generally does not need to return to real-address mode. To run software written to run in real-address mode (8086 mode), it is generally more convenient to run the software in virtual-8086 mode, than to switch back to real-address mode.

11.9.1 Switching to Protected Mode

Before switching to protected mode from real mode, a minimum set of system data structures and code modules must be loaded into memory, as described in Section 11.8, "Software Initialization for Protected-Mode Operation." Once these tables are created, software initialization code can switch into protected mode.

Protected mode is entered by executing a MOV CR0 instruction that sets the PE flag in the CR0 register. (In the same instruction, the PG flag in register CR0 can be set to enable paging.) Execution in protected mode begins with a CPL of 0.

Intel 64 and IA-32 processors have slightly different requirements for switching to protected mode. To ensure upwards and downwards code compatibility with Intel 64 and IA-32 processors, we recommend that you follow these steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry. (Software must guarantee that no exceptions or interrupts are generated during the mode switching operation.)
2. Execute the LGDT instruction to load the GDTR register with the base address of the GDT.
3. Execute a MOV CR0 instruction that sets the PE flag (and optionally the PG flag) in control register CR0.
4. Immediately following the MOV CR0 instruction, execute a far JMP or far CALL instruction. (This operation is typically a far jump or call to the next instruction in the instruction stream.)
5. The JMP or CALL instruction immediately after the MOV CR0 instruction changes the flow of execution and serializes the processor.
6. If paging is enabled, the code for the MOV CR0 instruction and the JMP or CALL instruction must come from a page that is identity mapped (that is, the linear address before the jump is the same as the physical address after paging and protected mode is enabled). The target instruction for the JMP or CALL instruction does not need to be identity mapped.
7. If a local descriptor table is going to be used, execute the LLDT instruction to load the segment selector for the LDT in the LDTR register.
8. Execute the LTR instruction to load the task register with a segment selector to the initial protected-mode task or to a writable area of memory that can be used to store TSS information on a task switch.
9. After entering protected mode, the segment registers continue to hold the contents they had in real-address mode. The JMP or CALL instruction in step 4 resets the CS register. Perform one of the following operations to update the contents of the remaining segment registers.
   - — Reload segment registers DS, SS, ES, FS, and GS. If the ES, FS, and/or GS registers are not going to be used, load them with a null selector.
   - — Perform a JMP or CALL instruction to a new task, which automatically resets the values of the segment registers and branches to a new code segment.
10. Execute the LIDT instruction to load the IDTR register with the address and limit of the protected-mode IDT.
11. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

Random failures can occur if other instructions exist between steps 3 and 4 above. Failures will be readily seen in some situations, such as when instructions that reference memory are inserted between steps 3 and 4 while in system management mode.

11.9.2 Switching Back to Real-Address Mode

The processor switches from protected mode back to real-address mode if software clears the PE bit in the CR0 register with a MOV CR0 instruction. A procedure that re-enters real-address mode should perform the following steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry.
2. If paging is enabled, perform the following operations:
   - Transfer program control to linear addresses that are identity mapped to physical addresses (that is, linear addresses equal physical addresses).
   - Ensure that the GDT and IDT are in identity mapped pages.
   - Clear the PG bit in the CR0 register.
   - Move 0H into the CR3 register to flush the TLB.
3. Transfer program control to a readable segment that has a limit of 64 KBytes (FFFFH). This operation loads the CS register with the segment limit required in real-address mode.
4. Load segment registers SS, DS, ES, FS, and GS with a selector for a descriptor containing the following values, which are appropriate for real-address mode:
   - Limit = 64 KBytes (0FFFFH)
   - Byte granular (G = 0)
   - Expand up (E = 0)
   - Writable (W = 1)
   - Present (P = 1)
   - Base = any value

   The segment registers must be loaded with non-null segment selectors or the segment registers will be unusable in real-address mode. Note that if the segment registers are not reloaded, execution continues using the descriptor attributes loaded during protected mode.
5. Execute an LIDT instruction to point to a real-address mode interrupt table that is within the 1-MByte real-address mode address range.
6. Clear the PE flag in the CR0 register to switch to real-address mode.
7. Execute a far JMP instruction to jump to a real-address mode program. This operation flushes the instruction queue and loads the appropriate base-address value in the CS register.
8. Load the SS, DS, ES, FS, and GS registers as needed by the real-address mode code. If any of the registers are not going to be used in real-address mode, write 0s to them.
9. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

NOTE

All the code that is executed in steps 1 through 9 must be in a single page and the linear addresses in that page must be identity mapped to physical addresses.

11.10 INITIALIZATION AND MODE SWITCHING EXAMPLE

This section provides an initialization and mode switching example that can be incorporated into an application. This code was originally written to initialize the Intel386 processor, but it will execute successfully on the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The code in this example is intended to reside in EPROM and to run following a hardware reset of the processor. The function of the code is to do the following:

- Establish a basic real-address mode operating environment.

- Load the necessary protected-mode system data structures into RAM.
- Load the system registers with the necessary pointers to the data structures and the appropriate flag settings for protected-mode operation.
- Switch the processor to protected mode.

Figure 11-3 shows the physical memory layout for the processor following a hardware reset and the starting point of this example. The EPROM that contains the initialization code resides at the upper end of the processor's physical memory address range, starting at address FFFFFFFFH and going down from there. The address of the first instruction to be executed is at FFFFFFF0H, the default starting address for the processor following a hardware reset.

The main steps carried out in this example are summarized in Table 11-5. The source listing for the example (with the filename STARTUP.ASM) is given in Example 11-1. The line numbers given in Table 11-5 refer to the source listing.

The following are some additional notes concerning this example:

- When the processor is switched into protected mode, the original code segment base-address value of FFFF0000H (located in the hidden part of the CS register) is retained and execution continues from the current offset in the EIP register. The processor will thus continue to execute code in the EPROM until a far jump or call is made to a new code segment, at which time, the base address in the CS register will be changed.
- Maskable hardware interrupts are disabled after a hardware reset and should remain disabled until the necessary interrupt handlers have been installed. The NMI interrupt is not disabled following a reset. The NMI# pin must thus be inhibited from being asserted until an NMI handler has been loaded and made available to the processor.
- The use of a temporary GDT allows simple transfer of tables from the EPROM to anywhere in the RAM area. A GDT entry is constructed with its base pointing to address 0 and a limit of 4 GBytes. When the DS and ES registers are loaded with this descriptor, the temporary GDT is no longer needed and can be replaced by the application GDT.
- This code loads one TSS and no LDTs. If more TSSs exist in the application, they must be loaded into RAM. If there are LDTs they may be loaded as well.

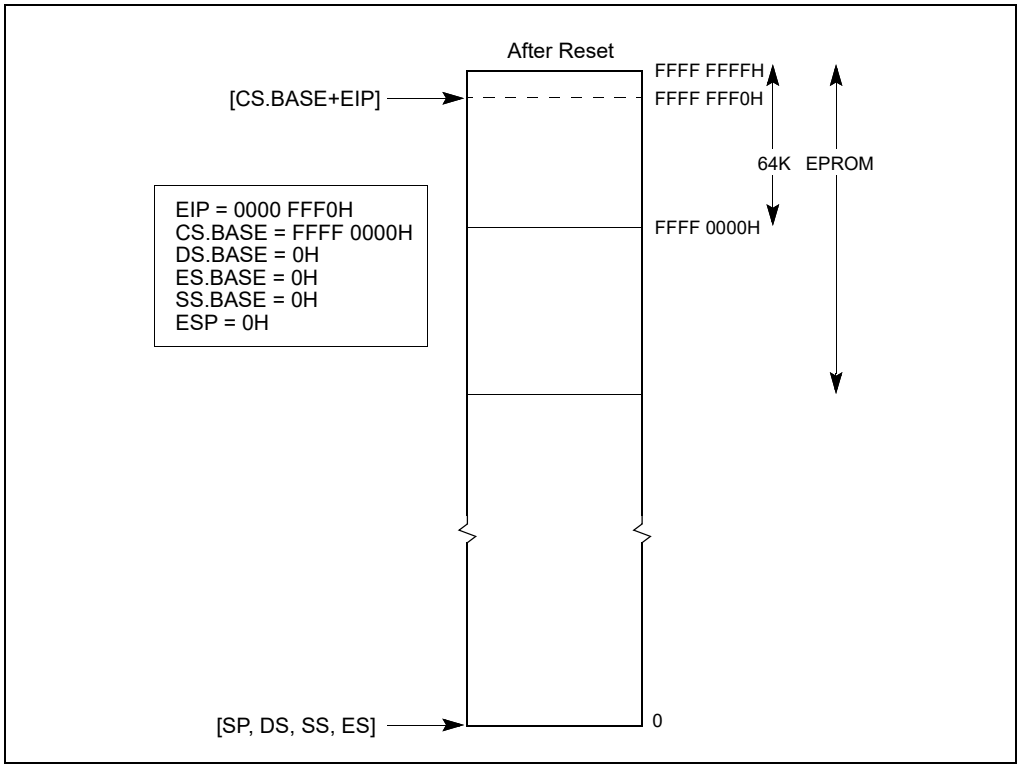


Figure 11-3. Processor State After Reset

Table 11-5. Main Initialization Steps in STARTUP.ASM Source Listing

STARTUP.ASM Line Numbers		Description
From	To	
157	157	Jump (short) to the entry code in the EPROM
162	169	Construct a temporary GDT in RAM with one entry: 0 - null 1 - R/W data segment, base = 0, limit = 4 GBytes
171	172	Load the GDTR to point to the temporary GDT
174	177	Load CR0 with PE flag set to switch to protected mode
179	181	Jump near to clear real mode instruction queue
184	186	Load DS, ES registers with GDT[1] descriptor, so both point to the entire physical memory space
188	195	Perform specific board initialization that is imposed by the new protected mode
196	218	Copy the application's GDT from ROM into RAM
220	238	Copy the application's IDT from ROM into RAM
241	243	Load application's GDTR
244	245	Load application's IDTR
247	261	Copy the application's TSS from ROM into RAM
263	267	Update TSS descriptor and other aliases in GDT (GDT alias or IDT alias)
277	277	Load the task register (without task switch) using LTR instruction
282	286	Load SS, ESP with the value found in the application's TSS
287	287	Push EFLAGS value found in the application's TSS
288	288	Push CS value found in the application's TSS
289	289	Push EIP value found in the application's TSS
290	293	Load DS, ES with the value found in the application's TSS
296	296	Perform IRET; pop the above values and enter the application code

11.10.1 Assembler Usage

In this example, the Intel assembler ASM386 and build tools BLD386 are used to assemble and build the initialization code module. The following assumptions are used when using the Intel ASM386 and BLD386 tools.

- The ASM386 will generate the right operand size opcodes according to the code-segment attribute. The attribute is assigned either by the ASM386 invocation controls or in the code-segment definition.
- If a code segment that is going to run in real-address mode is defined, it must be set to a USE 16 attribute. If a 32-bit operand is used in an instruction in this code segment (for example, MOV EAX, EBX), the assembler automatically generates an operand prefix for the instruction that forces the processor to execute a 32-bit operation, even though its default code-segment attribute is 16-bit.
- Intel's ASM386 assembler allows specific use of the 16- or 32-bit instructions, for example, LGDTW, LGDTD, IRETD. If the generic instruction LGDT is used, the default- segment attribute will be used to generate the right opcode.

11.10.2 STARTUP.ASM Listing

Example 11-1 provides high-level sample code designed to move the processor into protected mode. This listing does not include any opcode and offset information.

Example 11-1. STARTUP.ASM

```
MS-DOS* 5.0(045-N) 386(TM) MACRO ASSEMBLER STARTUP  09:44:51 08/19/92 PAGE 1

MS-DOS 5.0(045-N) 386(TM) MACRO ASSEMBLER V4.0, ASSEMBLY OF MODULE STARTUP
OBJECT MODULE PLACED IN startup.obj
ASSEMBLER INVOKED BY: f:\386tools\ASM386.EXE startup.a58 pw (132 )

LINE     SOURCE

   1      NAME    STARTUP
   2
   3  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   4  ;
   5  ;   ASSUMPTIONS:
   6  ;
   7  ;      1.  The bottom 64K of memory is ram, and can be used for
   8  ;          scratch space by this module.
   9  ;
  10  ;      2.  The system has sufficient free usable ram to copy the
  11  ;          initial GDT, IDT, and TSS
  12  ;
  13  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  14
  15  ; configuration data - must match with build definition
  16
  17  CS_BASE       EQU     0FFFF0000H
  18
  19   ; CS_BASE is the linear address of the segment STARTUP_CODE
  20   ; - this is specified in the build language file
  21
  22  RAM_START     EQU     400H
  23
  24  ; RAM_START  is the start of free, usable ram in the linear
  25  ; memory  space.   The GDT,  IDT, and  initial TSS  will be
  26  ; copied above this space, and a small data segment will be
  27  ; discarded at  this linear  address.   The 32-bit  word at
  28  ; RAM_START will contain  the linear  address of  the first
  29  ; free byte above the copied tables - this may be useful if
  30  ; a memory manager is used.
  31
  32  TSS_INDEX    EQU     10
  33
  34  ; TSS_INDEX is the  index of the  TSS of the  first task to
  35  ; run after startup
  36
  37
  38   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  39
  40  ; ------------------------- STRUCTURES and EQU ---------------
  41  ; structures for system data
  42
  43  ; TSS structure
  44  TASK_STATE  STRUC
  45      link     DW ?
```

```
46      link_h    DW ?
47      ESP0      DD ?
48      SS0       DW ?
49      SS0_h     DW ?
50      ESP1      DD ?
51      SS1       DW ?
52      SS1_h     DW ?
53      ESP2      DD ?
54      SS2       DW ?
55      SS2_h     DW ?
56      CR3_reg   DD ?
57      EIP_reg   DD ?
58      EFLAGS_regDD ?
59      EAX_reg   DD ?
60      ECX_reg   DD ?
61      EDX_reg   DD ?
62      EBX_reg   DD ?
63      ESP_reg   DD ?
64      EBP_reg   DD ?
65      ESI_reg   DD ?
66      EDI_reg   DD ?
67      ES_reg    DW ?
68      ES_h      DW ?
69      CS_reg    DW ?
70      CS_h      DW ?
71      SS_reg    DW ?
72      SS_h      DW ?
73      DS_reg    DW ?
74      DS_h      DW ?
75      FS_reg    DW ?
76      FS_h      DW ?
77      GS_reg    DW ?
78      GS_h      DW ?
79      LDT_reg   DW ?
80      LDT_h     DW ?
81      TRAP_reg  DW ?
82      IO_map_baseDW ?
83  TASK_STATE   ENDS
84
85  ; basic structure of a descriptor
86  DESC    STRUC
87      lim_0_15  DW ?
88      bas_0_15  DW ?
89      bas_16_23 DB ?
90      access    DB ?
91      gran      DB ?
92      bas_24_31 DB ?
93  DESC    ENDS
94
95  ; structure for use with LGDT and LIDT instructions
96  TABLE_REG   STRUC
97      table_limDW ?
98      table_linearDD ?
99  TABLE_REG   ENDS
```

```
100
101 ; offset of GDT and IDT descriptors in builder generated GDT
102 GDT_DESC_OFF    EQU 1*SIZE(DESC)
103 IDT_DESC_OFF    EQU 2*SIZE(DESC)
104
105 ; equates for building temporary GDT in RAM
106 LINEAR_SEL          EQU     1*SIZE (DESC)
107 LINEAR_PROTO_LO     EQU     00000FFFFH  ; LINEAR_ALIAS
108 LINEAR_PROTO_HI     EQU     000CF9200H
109
110 ; Protection Enable Bit in CR0
111 PE_BIT  EQU 1B
112
113 ; -------------------------------------------------------------
114
115 ; ------------------------- DATA SEGMENT----------------------
116
117 ; Initially, this  data segment starts at linear 0, according
118 ; to the processor's power-up state.
119
120 STARTUP_DATA    SEGMENT RW
121
122 free_mem_linear_base    LABEL   DWORD
123 TEMP_GDT                LABEL   BYTE  ; must be first in segment
124 TEMP_GDT_NULL_DESC   DESC    <>
125 TEMP_GDT_LINEAR_DESC DESC    <>
126
127 ; scratch areas for LGDT and LIDT instructions
128 TEMP_GDT_SCRATCH TABLE_REG   <>
129 APP_GDT_RAM     TABLE_REG    <>
130 APP_IDT_RAM     TABLE_REG    <>
131         ; align end_data
132 fill    DW      ?
133
134 ; last thing in this segment - should be on a dword boundary
135 end_data    LABEL   BYTE
136
137 STARTUP_DATA    ENDS
138 ; -------------------------------------------------------------
139
140
141 ; ------------------------- CODE SEGMENT----------------------
142 STARTUP_CODE SEGMENT ER PUBLIC USE16
143
144 ; filled in by builder
145     PUBLIC  GDT_EPROM
146 GDT_EPROM   TABLE_REG   <>
147
148 ; filled in by builder
149     PUBLIC  IDT_EPROM
150 IDT_EPROM   TABLE_REG   <>
151
152 ; entry point into startup code - the bootstrap will vector
153 ; here  with a  near JMP  generated by  the builder.   This
```

```
154  ; label must be in the top 64K of linear memory.
155
156      PUBLIC  STARTUP
157  STARTUP:
158
159  ; DS,ES address the bottom 64K of flat linear memory
160      ASSUME  DS:STARTUP_DATA, ES:STARTUP_DATA
161  ; See Figure 11-4
162  ; load GDTR with temporary GDT
163          LEA     EBX,TEMP_GDT  ; build the TEMP_GDT in low ram,
164          MOV     DWORD PTR [EBX],0   ; where we can address
165          MOV     DWORD PTR [EBX]+4,0
166          MOV     DWORD PTR [EBX]+8, LINEAR_PROTO_LO
167          MOV     DWORD PTR [EBX]+12, LINEAR_PROTO_HI
168          MOV     TEMP_GDT_scratch.table_linear,EBX
169          MOV     TEMP_GDT_scratch.table_lim,15
170
171          DB 66H; execute a 32 bit LGDT
172          LGDT    TEMP_GDT_scratch
173
174  ; enter protected mode
175          MOV     EBX,CR0
176          OR      EBX,PE_BIT
177          MOV     CR0,EBX
178

179   ; clear prefetch queue
180          JMP     CLEAR_LABEL
181  CLEAR_LABEL:
182
183   ; make DS and ES address 4G of linear memory
184          MOV     CX,LINEAR_SEL
185          MOV     DS,CX
186          MOV     ES,CX
187
188    ; do board specific initialization
189    ;
190                  ;
191                  ; ......
192                  ;
193
194
195          ; See Figure 11-5
196          ; copy EPROM GDT to ram at:
197          ;                RAM_START + size (STARTUP_DATA)
198          MOV     EAX,RAM_START
199          ADD     EAX,OFFSET (end_data)
200          MOV     EBX,RAM_START
201          MOV     ECX, CS_BASE
202          ADD     ECX, OFFSET (GDT_EPROM)
203          MOV     ESI, [ECX].table_linear
204          MOV     EDI,EAX
205          MOVZX   ECX, [ECX].table_lim
206          MOV     APP_GDT_ram[EBX].table_lim,CX
```

```
207          INC     ECX
208          MOV     EDX,EAX
209          MOV     APP_GDT_ram[EBX].table_linear,EAX
210          ADD     EAX,ECX
211      REP MOVS    BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
212
213          ; fixup GDT base in descriptor
214          MOV     ECX,EDX
215          MOV     [EDX].bas_0_15+GDT_DESC_OFF,CX
216          ROR     ECX,16
217          MOV     [EDX].bas_16_23+GDT_DESC_OFF,CL
218          MOV     [EDX].bas_24_31+GDT_DESC_OFF,CH
219
220          ; copy EPROM IDT to ram at:
221          ; RAM_START+size(STARTUP_DATA)+SIZE (EPROM GDT)
222          MOV     ECX, CS_BASE
223          ADD     ECX, OFFSET (IDT_EPROM)
224          MOV     ESI, [ECX].table_linear
225          MOV     EDI,EAX
226          MOVZX   ECX, [ECX].table_lim
227          MOV     APP_IDT_ram[EBX].table_lim,CX
228          INC     ECX
229          MOV     APP_IDT_ram[EBX].table_linear,EAX
230          MOV     EBX,EAX
231          ADD     EAX,ECX
232      REP MOVS    BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
233
234                  ; fixup IDT pointer in GDT
235          MOV     [EDX].bas_0_15+IDT_DESC_OFF,BX
236          ROR     EBX,16
237          MOV     [EDX].bas_16_23+IDT_DESC_OFF,BL
238          MOV     [EDX].bas_24_31+IDT_DESC_OFF,BH
239
240                  ; load GDTR and IDTR
241          MOV     EBX,RAM_START
242                  DB      66H         ; execute a 32 bit LGDT
243          LGDT    APP_GDT_ram[EBX]
244                  DB      66H         ; execute a 32 bit LIDT
245          LIDT    APP_IDT_ram[EBX]
246
247                  ; move the TSS
248          MOV     EDI,EAX
249          MOV     EBX,TSS_INDEX*SIZE(DESC)
250          MOV     ECX,GDT_DESC_OFF ;build linear address for TSS
251          MOV     GS,CX
252          MOV     DH,GS:[EBX].bas_24_31
253          MOV     DL,GS:[EBX].bas_16_23
254          ROL     EDX,16
255          MOV     DX,GS:[EBX].bas_0_15
256          MOV     ESI,EDX
257          LSL     ECX,EBX
258          INC     ECX
259          MOV     EDX,EAX
260          ADD     EAX,ECX
```

```
 261      REP MOVS    BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
 262
 263                  ; fixup TSS pointer
 264          MOV     GS:[EBX].bas_0_15,DX
 265          ROL     EDX,16
 266          MOV     GS:[EBX].bas_24_31,DH
 267          MOV     GS:[EBX].bas_16_23,DL
 268          ROL     EDX,16
 269      ;save start of free ram at linear location RAMSTART
 270          MOV     free_mem_linear_base+RAM_START,EAX
 271
 272      ;assume no  LDT used in  the initial task  - if necessary,
 273      ;code  to move the LDT could be added, and should resemble
 274      ;that used to move the TSS
 275
 276      ; load task register
 277          LTR     BX   ; No task switch, only descriptor loading
 278      ; See Figure 11-6
 279      ; load minimal set of registers necessary to simulate task
 280      ; switch
 281
 282
 283          MOV     AX,[EDX].SS_reg     ; start loading registers
 284          MOV     EDI,[EDX].ESP_reg
 285          MOV     SS,AX
 286          MOV     ESP,EDI             ; stack now valid
 287          PUSH    DWORD PTR [EDX].EFLAGS_reg
 288          PUSH    DWORD PTR [EDX].CS_reg
 289          PUSH    DWORD PTR [EDX].EIP_reg
 290          MOV     AX,[EDX].DS_reg
 291          MOV     BX,[EDX].ES_reg
 292          MOV     DS,AX     ; DS and ES no longer linear memory
 293          MOV     ES,BX
 294
 295          ; simulate far jump to initial task
 296          IRETD
 297
 298  STARTUP_CODE  ENDS
*** WARNING #377 IN 298, (PASS 2) SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)
 299
 300  END STARTUP, DS:STARTUP_DATA, SS:STARTUP_DATA
 301
 302

ASSEMBLY COMPLETE,   1 WARNING,   NO ERRORS.
```

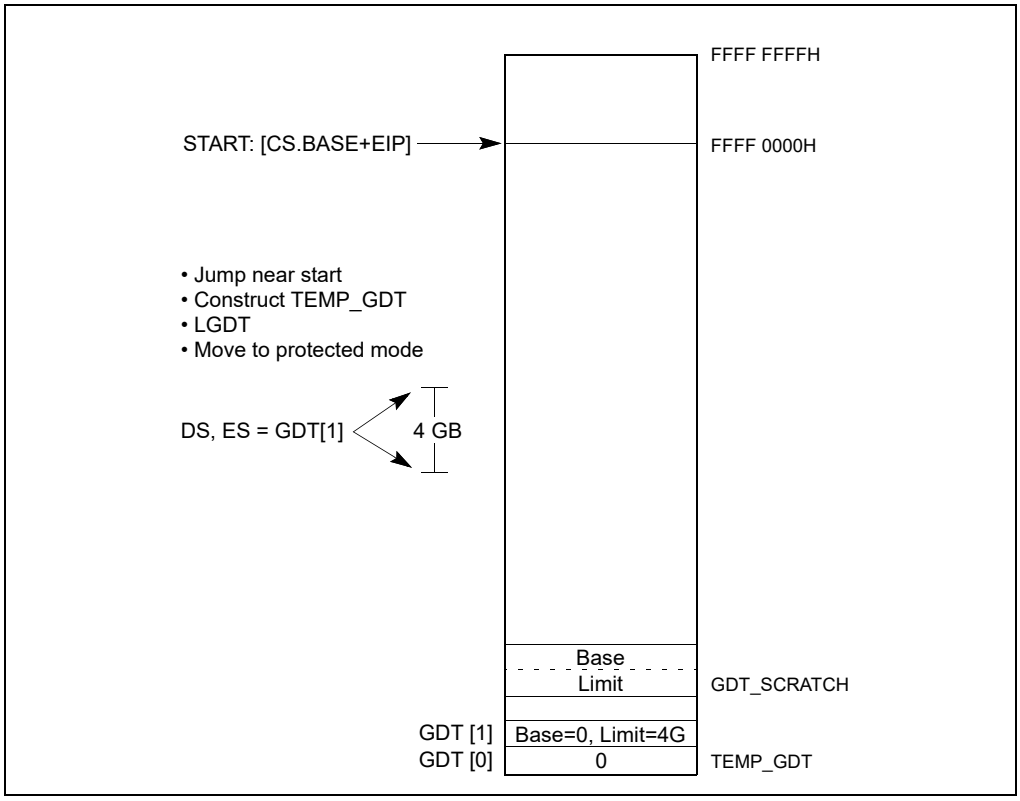


Figure 11-4. Constructing Temporary GDT and Switching to Protected Mode (Lines 162-172 of List File)

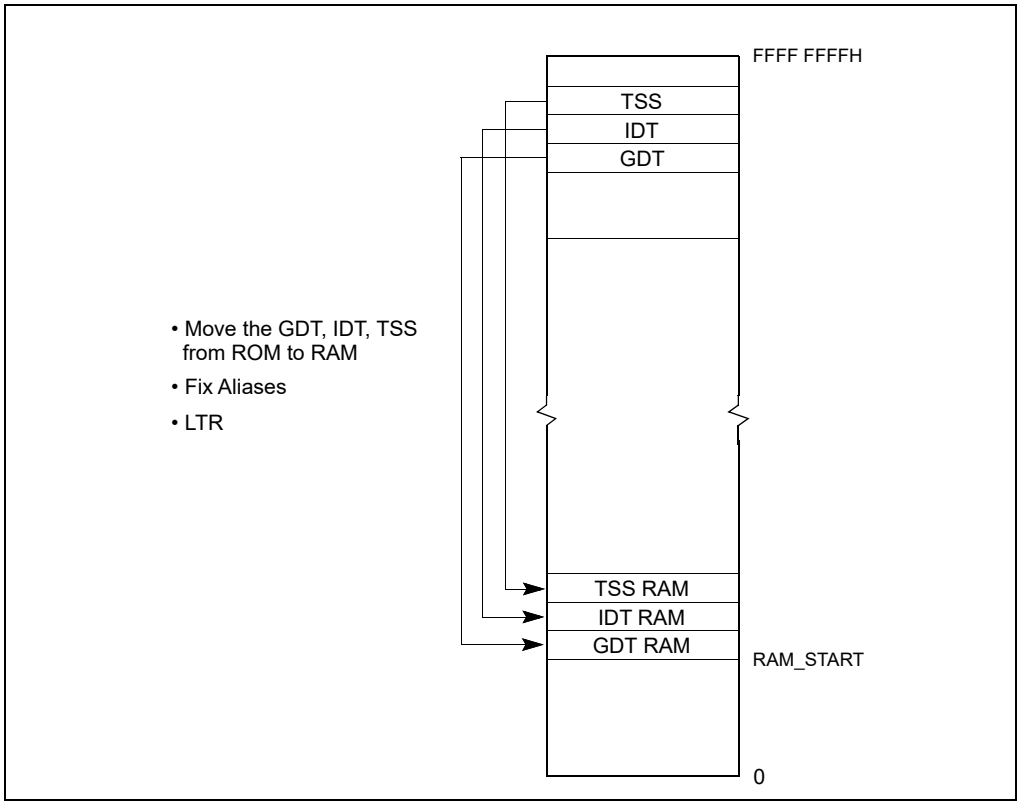


Figure 11-5. Moving the GDT, IDT, and TSS from ROM to RAM (Lines 196-261 of List File)

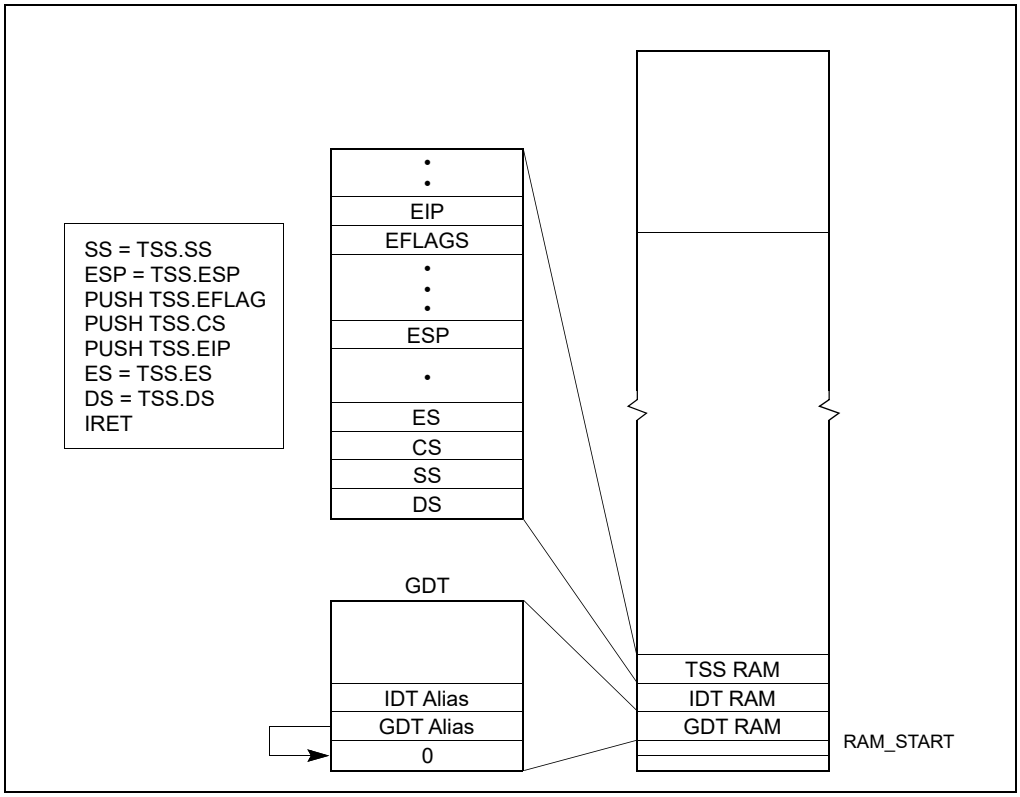


Figure 11-6. Task Switching (Lines 282-296 of List File)

11.10.3 MAIN.ASM Source Code

The file MAIN.ASM shown in Example 11-2 defines the data and stack segments for this application and can be substituted with the main module task written in a high-level language that is invoked by the IRET instruction executed by STARTUP.ASM.

Example 11-2. MAIN.ASM

```
NAME    main_module
data    SEGMENT RW
   dw 1000 dup(?)
DATA    ENDS
stack stackseg 800
CODE SEGMENT ER  use32 PUBLIC
main_start:
   nop
   nop
   nop
CODE  ENDS
END main_start, ds:data, ss:stack
```

11.10.4 Supporting Files

The batch file shown in Example 11-3 can be used to assemble the source code files STARTUP.ASM and MAIN.ASM and build the final application.

Example 11-3. Batch File to Assemble and Build the Application

```
ASM386 STARTUP.ASM
ASM386 MAIN.ASM
BLD386 STARTUP.OBJ, MAIN.OBJ buildfile(EPROM.BLD) bootstrap(STARTUP) Bootload

BLD386 performs several operations in this example:
It allocates physical memory location to segments and tables.
It generates tables using the build file and the input files.
It links object files and resolves references.
It generates a boot-loadable file to be programmed into the EPROM.
```

Example 11-4 shows the build file used as an input to BLD386 to perform the above functions.

Example 11-4. Build File

```
INIT_BLD_EXAMPLE;

SEGMENT
        *SEGMENTS(DPL = 0)
    ,   startup.startup_code(BASE = 0FFFF0000H)
    ;

TASK
        BOOT_TASK(OBJECT = startup, INITIAL,DPL = 0,
                    NOT INTENABLED)
,       PROTECTED_MODE_TASK(OBJECT = main_module,DPL = 0,
                    NOT INTENABLED)
    ;

TABLE
    GDT (
        LOCATION = GDT_EPROM
    ,   ENTRY = (
            10:   PROTECTED_MODE_TASK
    ,        startup.startup_code
    ,       startup.startup_data
    ,       main_module.data
    ,       main_module.code
    ,       main_module.stack
            )
        ),

    IDT (
        LOCATION = IDT_EPROM
        );

MEMORY
    (
        RESERVE = (0..3FFFH
                -- Area for the GDT, IDT, TSS copied from ROM
    ,              60000H..0FFFEFFFFH)
    ,   RANGE = (ROM_AREA = ROM (0FFFF0000H..0FFFFFFFFH))
                    -- Eprom size 64K
    ,   RANGE = (RAM_AREA = RAM (4000H..05FFFFH))
```

```
    );

END
```

Table 11-6 shows the relationship of each build item with an ASM source file.

Table 11-6. Relationship Between BLD Item and ASM Source File

Item	ASM386 and Startup.A58	BLD386 Controls and BLD file	Effect
Bootstrap	public startup startup:	bootstrap start(startup)	Near jump at 0FFFFFFF0H to start.
GDT location	public GDT_EPROM GDT_EPROM TABLE_REG <>	TABLE GDT(location = GDT_EPROM)	The location of the GDT will be programmed into the GDT_EPROM location.
IDT location	public IDT_EPROM IDT_EPROM TABLE_REG <>	TABLE IDT(location = IDT_EPROM	The location of the IDT will be programmed into the IDT_EPROM location.
RAM start	RAM_START equ 400H	memory (reserve = (0..3FFFH))	RAM_START is used as the ram destination for moving the tables. It must be excluded from the application's segment area.
Location of the application TSS in the GDT	TSS_INDEX EQU 10	TABLE GDT(ENTRY = (10: PROTECTED_MODE_ TASK))	Put the descriptor of the application TSS in GDT entry 10.
EPROM size and location	size and location of the initialization code	SEGMENT startup.code (base = 0FFFF0000H) ...memory (RANGE(ROM_AREA = ROM(x..y))	Initialization code size must be less than 64K and resides at upper most 64K of the 4-GByte memory space.

11.11 MICROCODE UPDATE FACILITIES

The P6 family and later processors have the capability to correct errata by loading an Intel-supplied data block into the processor. The data block is called a microcode update. This section describes the mechanisms the BIOS needs to provide in order to use this feature during system initialization. It also describes a specification that permits the incorporation of future updates into a system BIOS.

Intel considers the release of a microcode update for a silicon revision to be the equivalent of a processor stepping and completes a full-stepping level validation for releases of microcode updates.

A microcode update is used to correct errata in the processor. The BIOS, which has an update loader, is responsible for loading the update on processors during system initialization (Figure 11-7). There are two steps to this process: the first is to incorporate the necessary update data blocks into the BIOS; the second is to load update data blocks into the processor.

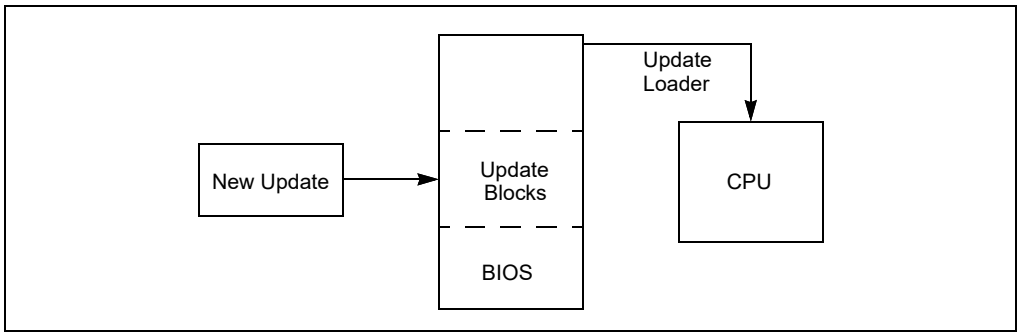


Figure 11-7. Applying Microcode Updates

11.11.1 Microcode Update

A microcode update consists of an Intel-supplied binary that contains a descriptive header and data. No executable code resides within the update. Each microcode update is tailored for a specific list of processor signatures. A mismatch of the processor's signature with the signature contained in the update will result in a failure to load. A processor signature includes the extended family, extended model, type, family, model, and stepping of the processor (starting with processor family 0fH, model 03H, a given microcode update may be associated with one of multiple processor signatures; see Section 11.11.2 for details).

Microcode updates are composed of a multi-byte header, followed by encrypted data and then by an optional extended signature table. Table 11-7 provides a definition of the fields; Table 11-8 shows the format of an update.

The header is 48 bytes. The first 4 bytes of the header contain the header version. The update header and its reserved fields are interpreted by software based upon the header version. An encoding scheme guards against tampering and provides a means for determining the authenticity of any given update. For microcode updates with a data size field equal to 00000000H, the size of the microcode update is 2048 bytes. The first 48 bytes contain the microcode update header. The remaining 2000 bytes contain encrypted data.

For microcode updates with a data size not equal to 00000000H, the total size field specifies the size of the micro-code update. The first 48 bytes contain the microcode update header. The second part of the microcode update is the encrypted data. The data size field of the microcode update header specifies the encrypted data size, its value must be a multiple of the size of DWORD. The total size field of the microcode update header specifies the encrypted data size plus the header size; its value must be in multiples of 1024 bytes (1 KBytes). The optional extended signature table if implemented follows the encrypted data, and its size is calculated by (Total Size – (Data Size + 48)).

NOTE

The optional extended signature table is supported starting with processor family 0FH, model 03H.

Table 11-7. Microcode Update Field Definitions

Field Name	Offset (bytes)	Length (bytes)	Description
Header Version	0	4	Version number of the update header.
Update Revision	4	4	Unique version number for the update, the basis for the update signature provided by the processor to indicate the current update functioning within the processor. Used by the BIOS to authenticate the update and verify that the processor loads successfully. The value in this field cannot be used for processor stepping identification alone. This is a signed 32-bit number.
Date	8	4	Date of the update creation in binary format: mmddyyyy (e.g., 07/18/98 is 07181998H).

Table 11-7. Microcode Update Field Definitions (Contd.)

Field Name	Offset (bytes)	Length (bytes)	Description
Processor Signature	12	4	*Extended family, extended model, type, family, model*, and *stepping* of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, *type, family, model*, and *stepping* of the processor. Software should use the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Checksum	16	4	Checksum of Update Data and Header. Used to verify the integrity of the update header and data. Checksum is correct when the summation of all the DWORDs (including the extended Processor Signature Table) that comprise the microcode update result in 00000000H.
Loader Revision	20	4	Version number of the loader program needed to correctly load this update. The initial version is 00000001H.
Processor Flags	24	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. Software should use the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Data Size	28	4	Specifies the size of the encrypted data in bytes, and must be a multiple of DWORDs. If this value is 00000000H, then the microcode update encrypted data is 2000 bytes (or 500 DWORDs).
Total Size	32	4	Specifies the total size of the microcode update in bytes. It is the summation of the header size, the encrypted data size and the size of the optional extended signature table. This value is always a multiple of 1024.
Reserved	36	12	Reserved fields for future expansion.
Update Data	48	Data Size or 2000	Update data.
Extended Signature Count	Data Size + 48	4	Specifies the number of extended signature structures (Processor Signature[n], processor flags[n] and checksum[n]) that exist in this microcode update.
Extended Checksum	Data Size + 52	4	Checksum of update extended processor signature table. Used to verify the integrity of the extended processor signature table. Checksum is correct when the summation of the DWORDs that comprise the extended processor signature table results in 00000000H.
Reserved	Data Size + 56	12	Reserved fields.

Table 11-7. Microcode Update Field Definitions (Contd.)

Field Name	Offset (bytes)	Length (bytes)	Description
Processor Signature[n]	Data Size + 68 + (n * 12)	4	*Extended family, extended model, type, family, model*, and *stepping* of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, *type, family, model*, and *stepping* of the processor. Software should use the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Processor Flags[n]	Data Size + 72 + (n * 12)	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. Software should use the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Checksum[n]	Data Size + 76 + (n * 12)	4	Used by utility software to decompose a microcode update into multiple microcode updates where each of the new updates is constructed without the optional Extended Processor Signature Table. To calculate the Checksum, substitute the Primary Processor Signature entry and the Processor Flags entry with the corresponding Extended Patch entry. Delete the Extended Processor Signature Table entries. The Checksum is correct when the summation of all DWORDs that comprise the created Extended Processor Patch results in 00000000H.

Table 11-8. Microcode Update Format

31 24	16	8	0	Bytes
Header Version				0
Update Revision				4
Month: 8	Day: 8	Year: 16		8
Processor Signature (CPUID)				12
Res: 4 / Extended Family: 8 / Extended Mode: 4 / Reserved: 2 / Type: 2 / Family: 4 / Model: 4 / Stepping: 4				
Checksum				16
Loader Revision				20
Processor Flags				24
Reserved (24 bits)			P7 P6 P5 P4 P3 P2 P1 P0	
Data Size				28
Total Size				32
Reserved (12 Bytes)				36

Table 11-8. Microcode Update Format (Contd.)

31 24 16 8 0	Bytes
Update Data (Data Size bytes, or 2000 Bytes if Data Size = 00000000H)	48
Extended Signature Count 'n'	Data Size + 48
Extended Processor Signature Table Checksum	Data Size + 52
Reserved (12 Bytes)	Data Size + 56
Processor Signature[n]	Data Size + 68 + (n * 12)
Processor Flags[n]	Data Size + 72 + (n * 12)
Checksum[n]	Data Size + 76 + (n * 12)

11.11.2 Optional Extended Signature Table

The extended signature table is a structure that may be appended to the end of the encrypted data when the encrypted data only supports a single processor signature (optional case). The extended signature table will always be present when the encrypted data supports multiple processor steppings and/or models (required case).

The extended signature table consists of a 20-byte extended signature header structure, which contains the extended signature count, the extended processor signature table checksum, and 12 reserved bytes (Table 11-9). Following the extended signature header structure, the extended signature table contains 0-to-n extended processor signature structures.

Each processor signature structure consist of the processor signature, processor flags, and a checksum (Table 11-10).

The extended signature count in the extended signature header structure indicates the number of processor signature structures that exist in the extended signature table.

The extended processor signature table checksum is a checksum of all DWORDs that comprise the extended signature table. That includes the extended signature count, extended processor signature table checksum, 12 reserved bytes and the n processor signature structures. A valid extended signature table exists when the result of a DWORD checksum is 00000000H.

Table 11-9. Extended Processor Signature Table Header Structure

Extended Signature Count 'n'	Data Size + 48
Extended Processor Signature Table Checksum	Data Size + 52
Reserved (12 Bytes)	Data Size + 56

Table 11-10. Processor Signature Structure

Processor Signature[n]	Data Size + 68 + (n * 12)
Processor Flags[n]	Data Size + 72 + (n * 12)
Checksum[n]	Data Size + 76 + (n * 12)

11.11.3 Processor Identification

Each microcode update is designed to for a specific processor or set of processors. To determine the correct microcode update to load, software must ensure that one of the processor signatures embedded in the microcode update matches the 32-bit processor signature returned by the CPUID instruction when executed by the target processor with EAX = 1. Attempting to load a microcode update that does not match a processor signature embedded in the microcode update with the processor signature returned by CPUID will cause the BIOS to reject the update.

Example 11-5 shows how to check for a valid processor signature match between the processor and microcode update.

Example 11-5. Pseudo Code to Validate the Processor Signature

```
ProcessorSignature ← CPUID(1):EAX

If (Update.HeaderVersion = 00000001h)
{
       // first check the ProcessorSignature field
    If (ProcessorSignature = Update.ProcessorSignature)
       Success

       // if extended signature is present
    Else If (Update.TotalSize > (Update.DataSize + 48))
    {

       //
       // Assume the Data Size has been used to calculate the
       // location of Update.ProcessorSignature[0].
       //

       For (N ← 0; ((N < Update.ExtendedSignatureCount) AND
           (ProcessorSignature ≠ Update.ProcessorSignature[N])); N++);

           // if the loops ended when the iteration count is
           // less than the number of processor signatures in
           // the table, we have a match
       If (N < Update.ExtendedSignatureCount)
           Success
       Else
           Fail
    }
    Else
       Fail
Else
    Fail
```

11.11.4 Platform Identification

In addition to verifying the processor signature, the intended processor platform type must be determined to properly target the microcode update. The intended processor platform type is determined by reading the IA32_PLATFORM_ID register, (MSR 17H). This 64-bit register must be read using the RDMSR instruction.

The three platform ID bits, when read as a binary coded decimal (BCD) number, indicate the bit position in the microcode update header's processor flags field associated with the installed processor. The processor flags in the 48-byte header and the processor flags field associated with the extended processor signature structures may have multiple bits set. Each set bit represents a different platform ID that the update supports.

Register Name: IA32_PLATFORM_ID
MSR Address: 017H

Access: Read Only

IA32_PLATFORM_ID is a 64-bit register accessed only when referenced as a Qword through a RDMSR instruction.

Table 11-11. Processor Flags

Bit	Descriptions
63:53	Reserved
52:50	Platform Id Bits (RO). The field gives information concerning the intended platform for the processor. See also Table 11-8. 52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7
49:0	Reserved

To validate the platform information, software may implement an algorithm similar to the algorithms in Example 11-6.

Example 11-6. Pseudo Code Example of Processor Flags Test

```
Flag ← 1 << IA32_PLATFORM_ID[52:50]

If (Update.HeaderVersion = 00000001h)
{
    If (Update.ProcessorFlags & Flag)
    {
        Load Update
    }
    Else
    {

        //
        // Assume the Data Size has been used to calculate the
        // location of Update.ProcessorSignature[N] and a match
        // on Update.ProcessorSignature[N] has already succeeded
        //

        If (Update.ProcessorFlags[n] & Flag)
        {
            Load Update
        }
    }
}
```

11.11.5 Microcode Update Checksum

Each microcode update contains a DWORD checksum located in the update header. It is software's responsibility to ensure that a microcode update is not corrupt. To check for a corrupt microcode update, software must perform a

unsigned DWORD (32-bit) checksum of the microcode update. Even though some fields are signed, the checksum procedure treats all DWORDs as unsigned. Microcode updates with a header version equal to 00000001H must sum all DWORDs that comprise the microcode update. A valid checksum check will yield a value of 00000000H. Any other value indicates the microcode update is corrupt and should not be loaded.

The checksum algorithm shown by the pseudo code in Example 11-7 treats the microcode update as an array of unsigned DWORDs. If the data size DWORD field at byte offset 32 equals 00000000H, the size of the encrypted data is 2000 bytes, resulting in 500 DWORDs. Otherwise the microcode update size in DWORDs = (*Total Size / 4*), where the total size is a multiple of 1024 bytes (1 KBytes).

Example 11-7. Pseudo Code Example of Checksum Test

```
N ← 512

If (Update.DataSize ≠ 00000000H)
   N ← Update.TotalSize / 4

ChkSum ← 0
For (I ← 0; I < N; I++)
{
   ChkSum ← ChkSum + MicrocodeUpdate[I]
}

If (ChkSum = 00000000H)
   Success
Else
   Fail
```

11.11.6 Microcode Update Loader

This section describes an update loader used to load an update into a P6 family or later processors. It also discusses the requirements placed on the BIOS to ensure proper loading. The update loader described contains the minimal instructions needed to load an update. The specific instruction sequence that is required to load an update is dependent upon the loader revision field contained within the update header. This revision is expected to change infrequently (potentially, only when new processor models are introduced).

Example 11-8 below represents the update loader with a loader revision of 00000001H. Note that the microcode update must be aligned on a 16-byte boundary and the size of the microcode update must be 1-KByte granular.

Example 11-8. Assembly Code Example of Simple Microcode Update Loader

```
mov  ecx,79h             ; MSR to write in ECX
xor  eax,eax             ; clear EAX
xor  ebx,ebx             ; clear EBX
mov  ax,cs               ; Segment of microcode update
shl  eax,4
mov  bx,offset Update    ; Offset of microcode update
add  eax,ebx             ; Linear Address of Update in EAX
add  eax,48d             ; Offset of the Update Data within the Update
xor  edx,edx             ; Zero in EDX
WRMSR                   ; microcode update trigger
```

The loader shown in Example 11-8 assumes that *update* is the address of a microcode update (header and data) embedded within the code segment of the BIOS. It also assumes that the processor is operating in real mode. The data may reside anywhere in memory, aligned on a 16-byte boundary, that is accessible by the processor within its current operating mode.

Before the BIOS executes the microcode update trigger (WRMSR) instruction, the following must be true:

- In 64-bit mode, EAX contains the lower 32-bits of the microcode update linear address. In protected mode, EAX contains the full 32-bit linear address of the microcode update.
- In 64-bit mode, EDX contains the upper 32-bits of the microcode update linear address. In protected mode, EDX equals zero.
- ECX contains 79H (address of IA32_BIOS_UPDT_TRIG).

Other requirements are:

- The addresses for the microcode update data must be in canonical form.
- If paging is enabled, the microcode update data must map that data as present.
- The microcode update data must start at a 16-byte aligned linear address.

11.11.6.1 Hard Resets in Update Loading

The effects of a loaded update are cleared from the processor upon a hard reset. Therefore, each time a hard reset is asserted during the BIOS POST, the update must be reloaded on all processors that observed the reset. The effects of a loaded update are, however, maintained across a processor INIT. There are no side effects caused by loading an update into a processor multiple times.

11.11.6.2 Update in a Multiprocessor System

A multiprocessor (MP) system requires loading each processor with update data appropriate for its CPUID and platform ID bits. The BIOS is responsible for ensuring that this requirement is met and that the loader is located in a module executed by all processors in the system. If a system design permits multiple steppings of Pentium 4, Intel Xeon, and P6 family processors to exist concurrently; then the BIOS must verify individual processors against the update header information to ensure appropriate loading. Given these considerations, it is most practical to load the update during MP initialization.

11.11.6.3 Update in a System Supporting Intel Hyper-Threading Technology

Intel Hyper-Threading Technology has implications on the loading of the microcode update. The update must be loaded for each core in a physical processor. Thus, for a processor supporting Intel Hyper-Threading Technology, only one logical processor per core is required to load the microcode update. Each individual logical processor can independently load the update. However, MP initialization must provide some mechanism (e.g., a software semaphore) to force serialization of microcode update loads and to prevent simultaneous load attempts to the same core.

11.11.6.4 Update in a System Supporting Dual-Core Technology

Dual-core technology has implications on the loading of the microcode update. The microcode update facility is not shared between processor cores in the same physical package. The update must be loaded for each core in a physical processor.

If processor core supports Intel Hyper-Threading Technology, the guideline described in Section 11.11.6.3 also applies.

11.11.6.5 Update Loader Enhancements

The update loader presented in Section 11.11.6, "Microcode Update Loader," is a minimal implementation that can be enhanced to provide additional functionality. Potential enhancements are described below:

- BIOS can incorporate multiple updates to support multiple steppings of the Pentium 4, Intel Xeon, and P6 family processors. This feature provides for operating in a mixed stepping environment on an MP system and enables a user to upgrade to a later version of the processor. In this case, modify the loader to check the CPUID and platform ID bits of the processor that it is running on against the available headers before loading a particular update. The number of updates is only limited by available BIOS space.

- A loader can load the update and test the processor to determine if the update was loaded correctly. See Section 11.11.7, "Update Signature and Verification."
- A loader can verify the integrity of the update data by performing a checksum on the double words of the update summing to zero. See Section 11.11.5, "Microcode Update Checksum."
- A loader can provide power-on messages indicating successful loading of an update.

11.11.7 Update Signature and Verification

The P6 family and later processors provide capabilities to verify the authenticity of a particular update and to identify the current update revision. This section describes the model-specific extensions of processors that support this feature. The update verification method below assumes that the BIOS will only verify an update that is more recent than the revision currently loaded in the processor.

CPUID returns a value in a model specific register in addition to its usual register return values. The semantics of CPUID cause it to deposit an update ID value in the 64-bit model-specific register at address 08BH (IA32_BIOS_SIGN_ID). If no update is present in the processor, the value in the MSR remains unmodified. The BIOS must pre-load a zero into the MSR before executing CPUID. If a read of the MSR at 8BH still returns zero after executing CPUID, this indicates that no update is present.

The update ID value returned in the EDX register after RDMSR executes indicates the revision of the update loaded in the processor. This value, in combination with the CPUID value returned in the EAX register, uniquely identifies a particular update. The signature ID can be directly compared with the update revision field in a microcode update header for verification of a correct load. No consecutive updates released for a given stepping of a processor may share the same signature. The processor signature returned by CPUID differentiates updates for different steppings.

11.11.7.1 Determining the Signature

An update that is successfully loaded into the processor provides a signature that matches the update revision of the currently functioning revision. This signature is available any time after the actual update has been loaded. Requesting the signature does not have a negative impact upon a loaded update.

The procedure for determining this signature shown in Example 11-9.

Example 11-9. Assembly Code to Retrieve the Update Revision

```
MOV    ECX, 08BH          ;IA32_BIOS_SIGN_ID
XOR    EAX, EAX           ;clear EAX
XOR    EDX, EDX           ;clear EDX
WRMSR                     ;Load 0 to MSR at 8BH
MOV    EAX, 1
cpuid
MOV    ECX, 08BH           ;IA32_BIOS_SIGN_ID
rdmsr                      ;Read Model Specific Register
```

If there is an update active in the processor, its revision is returned in the EDX register after the RDMSR instruction executes.

IA32_BIOS_SIGN_ID	Microcode Update Signature Register
MSR Address:	08BH Accessed as a Qword
Default Value:	XXXX XXXX XXXX XXXXh
Access:	Read/Write

The IA32_BIOS_SIGN_ID register is used to report the microcode update signature when CPUID executes. The signature is returned in the upper DWORD (Table 11-12).

Table 11-12. Microcode Update Signature

Bit	Description
63:32	Microcode update signature. This field contains the signature of the currently loaded microcode update when read following the execution of the CPUID instruction, function 1. It is required that this register field be pre-loaded with zero prior to executing the CPUID, function 1. If the field remains equal to zero, then there is no microcode update loaded. Another non-zero value will be the signature.
31:0	Reserved.

11.11.7.2 Authenticating the Update

An update may be authenticated by the BIOS using the signature primitive, described above, and the algorithm in Example 11-10.

Example 11-10. Pseudo Code to Authenticate the Update

```
Z ← Obtain Update Revision from the Update Header to be authenticated;
X ← Obtain Current Update Signature from MSR 8BH;

If (Z > X)
{
    Load Update that is to be authenticated;
    Y ← Obtain New Signature from MSR 8BH;

    If (Z = Y)
        Success
    Else
        Fail
}
Else
    Fail
```

Example 11-10 requires that the BIOS only authenticate updates that contain a numerically larger revision than the currently loaded revision, where Current Signature (X) < New Update Revision (Z). A processor with no loaded update is considered to have a revision equal to zero.

This authentication procedure relies upon the decoding provided by the processor to verify an update from a potentially hostile source. As an example, this mechanism in conjunction with other safeguards provides security for dynamically incorporating field updates into the BIOS.

11.11.8 Optional Processor Microcode Update Specifications

This section an interface that an OEM-BIOS may provide to its client system software to manage processor microcode updates. System software may choose to build its own facility to manage microcode updates (e.g., similar to the facility described in Section 9.11.6) or rely on a facility provided by the BIOS to perform microcode updates.

Sections 11.11.8.1-11.11.8.9 describes an extension (Function 0D042H) to the real mode INT 15H service. INT 15H 0D042H function is one of several alternatives that a BIOS may choose to implement microcode update facility and offer to its client application (e.g., an OS). Other alternative microcode update facility that BIOS can choose are dependent on platform-specific capabilities, including the Capsule Update mechanism from the UEFI specification (www.uefi.org). In this discussion, the application is referred to as the calling program or caller.

The real mode INT15 call specification described here is an Intel extension to an OEM BIOS. This extension allows an application to read and modify the contents of the microcode update data in NVRAM. The update loader, which is part of the system BIOS, cannot be updated by the interface. All of the functions defined in the specification must be implemented for a system to be considered compliant with the specification. The INT15 functions are accessible only from real mode.

11.11.8.1 Responsibilities of the BIOS

If a BIOS passes the presence test (INT 15H, AX = 0D042H, BL = 0H), it must implement all of the sub-functions defined in the INT 15H, AX = 0D042H specification. There are no optional functions. BIOS must load the appropriate update for each processor during system initialization.

A Header Version of an update block containing the value 0FFFFFFFFH indicates that the update block is unused and available for storing a new update.

The BIOS is responsible for providing a region of non-volatile storage (NVRAM) for each potential processor stepping within a system. This storage unit consists of one or more update blocks. An update block is a contiguous 2048-byte block of memory. The BIOS for a single processor system need only provide update blocks to store one microcode update. If the BIOS for a multiple processor system is intended to support mixed processor steppings, then the BIOS needs to provide enough update blocks to store each unique microcode update or for each processor socket on the OEM's system board.

The BIOS is responsible for managing the NVRAM update blocks. This includes garbage collection, such as removing microcode updates that exist in NVRAM for which a corresponding processor does not exist in the system. This specification only provides the mechanism for ensuring security, the uniqueness of an entry, and that stale entries are not loaded. The actual update block management is implementation specific on a per-BIOS basis.

As an example, the BIOS may use update blocks sequentially in ascending order with CPU signatures sorted versus the first available block. In addition, garbage collection may be implemented as a setup option to clear all NVRAM slots or as BIOS code that searches and eliminates unused entries during boot.

NOTES

For IA-32 processors starting with family 0FH and model 03H and Intel 64 processors, the microcode update may be as large as 16 KBytes. Thus, BIOS must allocate 8 update blocks for each microcode update. In a MP system, a common microcode update may be sufficient for each socket in the system.

For IA-32 processors earlier than family 0FH and model 03H, the microcode update is 2 KBytes. An MP-capable BIOS that supports multiple steppings must allocate a block for each socket in the system.

A single-processor BIOS that supports variable-sized microcode update and fixed-sized microcode update must allocate one 16-KByte region and a second region of at least 2 KBytes.

The following algorithm (Example 11-11) describes the steps performed during BIOS initialization used to load the updates into the processor(s). The algorithm assumes:

- The BIOS ensures that no update contained within NVRAM has a header version or loader version that does not match one currently supported by the BIOS.
- The update contains a correct checksum.
- The BIOS ensures that (at most) one update exists for each processor stepping.
- Older update revisions are not allowed to overwrite more recent ones.

These requirements are checked by the BIOS during the execution of the write update function of this interface. The BIOS sequentially scans through all of the update blocks in NVRAM starting with index 0. The BIOS scans until it finds an update where the processor fields in the header match the processor signature (extended family, extended model, type, family, model, and stepping) as well as the platform bits of the current processor.

Example 11-11. Pseudo Code, Checks Required Prior to Loading an Update

```
For each processor in the system
{
    Determine the Processor Signature via CPUID function 1;
    Determine the Platform Bits ← 1 << IA32_PLATFORM_ID[52:50];

    For (I ← UpdateBlock 0, I < NumOfBlocks; I++)
    {
        If (Update.Header_Version = 00000001H)
        {
```

```
            If ((Update.ProcessorSignature = Processor Signature) &&
                 (Update.ProcessorFlags & Platform Bits))
            {
                Load Update.UpdateData into the Processor;
                Verify update was correctly loaded into the processor
                Go on to next processor
                    Break;
            }
            Else If (Update.TotalSize > (Update.DataSize + 48))
            {
                N ← 0
                While (N < Update.ExtendedSignatureCount)
                {
                    If ((Update.ProcessorSignature[N] =
                         Processor Signature) &&
                         (Update.ProcessorFlags[N] & Platform Bits))
                    {
                        Load Update.UpdateData into the Processor;
                        Verify update correctly loaded into the processor
                        Go on to next processor
                            Break;
                    }
                    N ← N + 1
                }
                I ← I + (Update.TotalSize / 2048)
                If ((Update.TotalSize MOD 2048) = 0)
                    I ← I + 1
            }
        }
    }
}
```

NOTES

The platform Id bits in IA32_PLATFORM_ID are encoded as a three-bit binary coded decimal field. The platform bits in the microcode update header are individually bit encoded. The algorithm must do a translation from one format to the other prior to doing a check.

When performing the INT 15H, 0D042H functions, the BIOS must assume that the caller has no knowledge of platform specific requirements. It is the responsibility of BIOS calls to manage all chipset and platform specific prerequisites for managing the NVRAM device. When writing the update data using the Write Update sub-function, the BIOS must maintain implementation specific data requirements (such as the update of NVRAM checksum). The BIOS should also attempt to verify the success of write operations on the storage device used to record the update.

11.11.8.2 Responsibilities of the Calling Program

This section of the document lists the responsibilities of a calling program using the interface specifications to load microcode update(s) into BIOS NVRAM.

- The calling program should call the INT 15H, 0D042H functions from a pure real mode program and should be executing on a system that is running in pure real mode.
- The caller should issue the presence test function (sub function 0) and verify the signature and return codes of that function.
- It is important that the calling program provides the required scratch RAM buffers for the BIOS and the proper stack size as specified in the interface definition.
- The calling program should read any update data that already exists in the BIOS in order to make decisions about the appropriateness of loading the update. The BIOS must refuse to overwrite a newer update with an

older version. The update header contains information about version and processor specifics for the calling program to make an intelligent decision about loading.

- There can be no ambiguous updates. The BIOS must refuse to allow multiple updates for the same CPU to exist at the same time; it also must refuse to load updates for processors that don't exist on the system.
- The calling application should implement a verify function that is run after the update write function successfully completes. This function reads back the update and verifies that the BIOS returned an image identical to the one that was written.

Example 11-12 represents a calling program.

Example 11-12. INT 15 DO42 Calling Program Pseudo-code

```
//
// We must be in real mode
//
If the system is not in Real mode exit
//
// Detect presence of Genuine Intel processor(s) that can be updated
// using(CPUID)
//
If no Intel processors exist that can be updated exit
//
// Detect the presence of the Intel microcode update extensions
//
If the BIOS fails the PresenceTestexit
//
// If the APIC is enabled, see if any other processors are out there
//
Read IA32_APICBASE
If APIC enabled
{
    Send Broadcast Message to all processors except self via APIC
    Have all processors execute CPUID, record the Processor Signature
    (i.e.,Extended Family, Extended Model, Type, Family, Model, Stepping)
    Have all processors read IA32_PLATFORM_ID[52:50], record Platform
     Id Bits

    If current processor cannot be updated
       exit
}
//
// Determine the number of unique update blocks needed for this system
//
NumBlocks = 0
For each processor
{
    If ((this is a unique processor stepping) AND
       (we have a unique update in the database for this processor))
    {
       Checksum the update from the database;
       If Checksum fails
          exit
       NumBlocks ← NumBlocks + size of microcode update / 2048
    }
}

//
// Do we have enough update slots for all CPUs?
//
```

```
If there are more blocks required to support the unique processor steppings than update blocks
provided by the BIOS exit
//
// Do we need any update blocks at all?  If not, we are done
//
If (NumBlocks = 0)
   exit
//
// Record updates for processors in NVRAM.
//
For (I=0; I<NumBlocks; I++)
{
   //
   // Load each Update
   //
   Issue the WriteUpdate function

   If (STORAGE_FULL) returned
   {
      Display Error -- BIOS is not managing NVRAM appropriately
      exit
   }

   If (INVALID_REVISION) returned
   {
      Display Message: More recent update already loaded in NVRAM for
       this stepping
      continue
   }

   If any other error returned
   {
      Display Diagnostic
      exit
   }

   //
   // Verify the update was loaded correctly
   //
   Issue the ReadUpdate function

   If an error occurred
   {
      Display Diagnostic
      exit
   }
   //
   // Compare the Update read to that written
   //
   If (Update read ≠ Update written)
   {
      Display Diagnostic
      exit
   }

   I ← I + (size of microcode update / 2048)
}
//
// Enable Update Loading, and inform user
```

```
//
Issue the Update Control function with Task = Enable.
```

11.11.8.3 Microcode Update Functions

Table 11-13 defines the processor microcode update functions that implementations of INT 15H 0D042H must support.

Table 11-13. Microcode Update Functions

Microcode Update Function	Function Number	Description	Required/Optional
Presence test	00H	Returns information about the supported functions.	Required
Write update data	01H	Writes one of the update data areas (slots).	Required
Update control	02H	Globally controls the loading of updates.	Required
Read update data	03H	Reads one of the update data areas (slots).	Required

11.11.8.4 INT 15H-based Interface

If an OEM-BIOS is implementing INT 15H 0D042H interface and offer to its client, the BIOS should allow additional microcode updates to be added to system flash.

The program that calls this interface is responsible for providing three 64-kilobyte RAM areas for BIOS use during calls to the read and write functions. These RAM scratch pads can be used by the BIOS for any purpose, but only for the duration of the function call. The calling routine places real mode segments pointing to the RAM blocks in the CX, DX, and SI registers. Calls to functions in this interface must be made with a minimum of 32 kilobytes of stack available to the BIOS.

In general, each function returns with CF cleared and AH contains the returned status. The general return codes and other constant definitions are listed in Section 11.11.8.9, "Return Codes."

The OEM error field (AL) is provided for the OEM to return additional error information specific to the platform. If the BIOS provides no additional information about the error, OEM error must be set to SUCCESS. The OEM error field is undefined if AH contains either SUCCESS (00H) or NOT_IMPLEMENTED (86H). In all other cases, it must be set with either SUCCESS or a value meaningful to the OEM.

The following sections describe functions provided by the INT15H-based interface.

11.11.8.5 Function 00H—Presence Test

This function verifies that the BIOS has implemented required microcode update functions. Table 11-14 lists the parameters and return codes for the function.

Table 11-14. Parameters for the Presence Test

Input		
AX	Function Code	0D042H
BL	Sub-function	00H - Presence test
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid
AH	Return Code	
AL	OEM Error	Additional OEM information.
EBX	Signature Part 1	'INTE' - Part one of the signature
ECX	Signature Part 2	'LPEP'- Part two of the signature
EDX	Loader Version	Version number of the microcode update loader

Table 11-14. Parameters for the Presence Test (Contd.)

Input		
SI	Update Count	Number of 2048 update blocks in NVRAM the BIOS allocated to storing microcode updates
Return Codes (see Table 11-19 for code definitions		
SUCCESS		The function completed successfully.
NOT_IMPLEMENTED		The function is not implemented.

In order to assure that the BIOS function is present, the caller must verify the carry flag, the return code, and the 64-bit signature. The update count reflects the number of 2048-byte blocks available for storage within one non-volatile RAM.

The loader version number refers to the revision of the update loader program that is included in the system BIOS image.

11.11.8.6 Function 01H—Write Microcode Update Data

This function integrates a new microcode update into the BIOS storage device. Table 11-15 lists the parameters and return codes for the function.

Table 11-15. Parameters for the Write Update Data Function

Input		
AX	Function Code	0D042H
BL	Sub-function	01H - Write update
ES:DI	Update Address	Real Mode pointer to the Intel Update structure. This buffer is 2048 bytes in length if the processor supports only fixed-size microcode update or... Real Mode pointer to the Intel Update structure. This buffer is 64 KBytes in length if the processor supports a variable-size microcode update.
CX	Scratch Pad1	Real mode segment address of 64 KBytes of RAM block
DX	Scratch Pad2	Real mode segment address of 64 KBytes of RAM block
SI	Scratch Pad3	Real mode segment address of 64 KBytes of RAM block
SS:SP	Stack pointer	32 KBytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH Contains status Carry Clear - All return values valid
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM information
Return Codes (see Table 11-19 for code definitions		
SUCCESS		The function completed successfully.
NOT_IMPLEMENTED		The function is not implemented.
WRITE_FAILURE		A failure occurred because of the inability to write the storage device.
ERASE_FAILURE		A failure occurred because of the inability to erase the storage device.
READ_FAILURE		A failure occurred because of the inability to read the storage device.

Table 11-15. Parameters for the Write Update Data Function (Contd.)

Input	
STORAGE_FULL	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT	The processor stepping does not currently exist in the system.
INVALID_HEADER	The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	The update does not checksum correctly.
SECURITY_FAILURE	The processor rejected the update.
INVALID_REVISION	The same or more recent revision of the update exists in the storage device.

Description

The BIOS is responsible for selecting an appropriate update block in the non-volatile storage for storing the new update. This BIOS is also responsible for ensuring the integrity of the information provided by the caller, including authenticating the proposed update before incorporating it into storage.

Before writing the update block into NVRAM, the BIOS should ensure that the update structure meets the following criteria in the following order:

1. The update header version should be equal to an update header version recognized by the BIOS.
2. The update loader version in the update header should be equal to the update loader version contained within the BIOS image.
3. The update block must checksum. This checksum is computed as a 32-bit summation of all double words in the structure, including the header, data, and processor signature table.

The BIOS selects update block(s) in non-volatile storage for storing the candidate update. The BIOS can select any available update block as long as it guarantees that only a single update exists for any given processor stepping in non-volatile storage. If the update block selected already contains an update, the following additional criteria apply to overwrite it:

- The processor signature in the proposed update must be equal to the processor signature in the header of the current update in NVRAM (Processor Signature + platform ID bits).
- The update revision in the proposed update should be greater than the update revision in the header of the current update in NVRAM.

If no unused update blocks are available and the above criteria are not met, the BIOS can overwrite update block(s) for a processor stepping that is no longer present in the system. This can be done by scanning the update blocks and comparing the processor steppings, identified in the MP Specification table, to the processor steppings that currently exist in the system.

Finally, before storing the proposed update in NVRAM, the BIOS must verify the authenticity of the update via the mechanism described in Section 11.11.6, "Microcode Update Loader." This includes loading the update into the current processor, executing the CPUID instruction, reading MSR 08Bh, and comparing a calculated value with the update revision in the proposed update header for equality.

When performing the write update function, the BIOS must record the entire update, including the header, the update data, and the extended processor signature table (if applicable). When writing an update, the original contents may be overwritten, assuming the above criteria have been met. It is the responsibility of the BIOS to ensure that more recent updates are not overwritten through the use of this BIOS call, and that only a single update exists within the NVRAM for any processor stepping and platform ID.

Figure 11-8 and Figure 11-9 show the process the BIOS follows to choose an update block and ensure the integrity of the data when it stores the new microcode update.

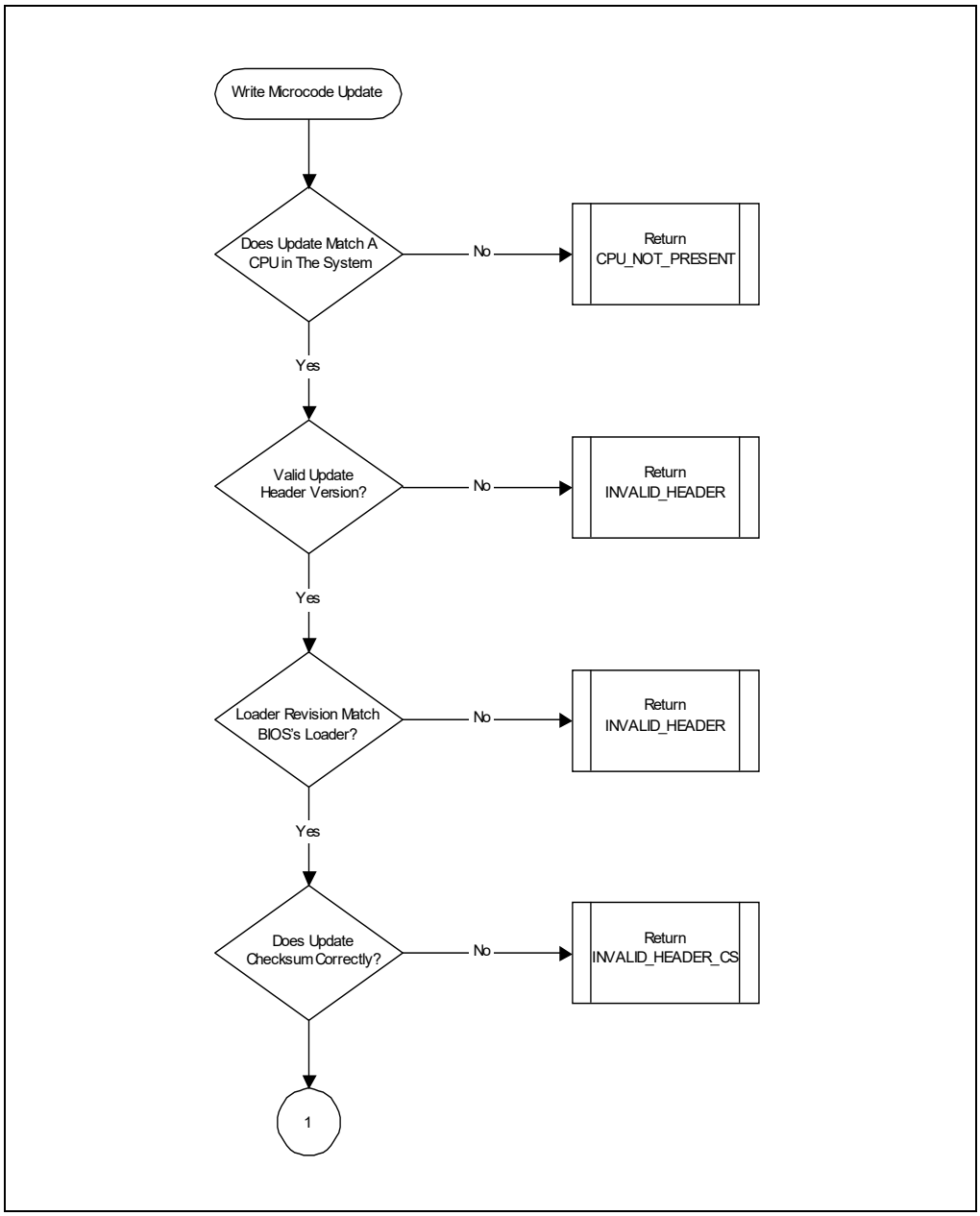


Figure 11-8. Microcode Update Write Operation Flow [1]

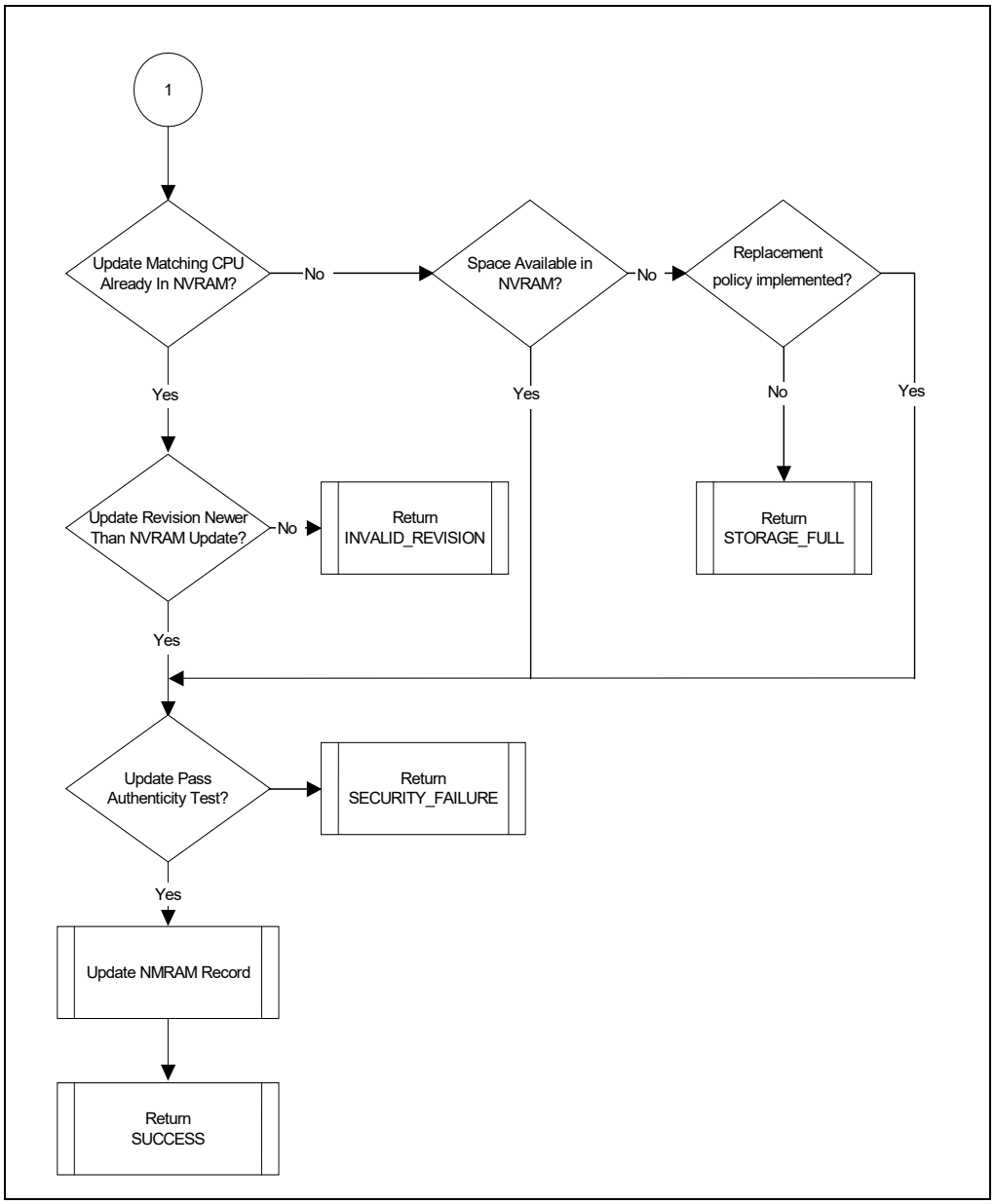


Figure 11-9. Microcode Update Write Operation Flow [2]

11.11.8.7 Function 02H—Microcode Update Control

This function enables loading of binary updates into the processor. Table 11-16 lists the parameters and return codes for the function.

Table 11-16. Parameters for the Control Update Sub-function

Input		
AX	Function Code	0D042H
BL	Sub-function	02H - Control update
BH	Task	See the description below.
CX	Scratch Pad1	Real mode segment of 64 KBytes of RAM block
DX	Scratch Pad2	Real mode segment of 64 KBytes of RAM block
SI	Scratch Pad3	Real mode segment of 64 KBytes of RAM block
SS:SP	Stack pointer	32 kilobytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid.
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM Information.
BL	Update Status	Either enable or disable indicator
Return Codes (see Table 11-19 for code definitions)		
SUCCESS		Function completed successfully.
READ_FAILURE		A failure occurred because of the inability to read the storage device.

This control is provided on a global basis for all updates and processors. The caller can determine the current status of update loading (enabled or disabled) without changing the state. The function does not allow the caller to disable loading of binary updates, as this poses a security risk.

The caller specifies the requested operation by placing one of the values from Table 11-17 in the BH register. After successfully completing this function, the BL register contains either the enable or the disable designator. Note that if the function fails, the update status return value is undefined.

Table 11-17. Mnemonic Values

Mnemonic	**Value**	**Meaning**
Enable	1	Enable the Update loading at initialization time.
Query	2	Determine the current state of the update control without changing its status.

The READ_FAILURE error code returned by this function has meaning only if the control function is implemented in the BIOS NVRAM. The state of this feature (enabled/disabled) can also be implemented using CMOS RAM bits where READ failure errors cannot occur.

11.11.8.8 Function 03H—Read Microcode Update Data

This function reads a currently installed microcode update from the BIOS storage into a caller-provided RAM buffer. Table 11-18 lists the parameters and return codes.

Table 11-18. Parameters for the Read Microcode Update Data Function

Input		
AX	Function Code	0D042H
BL	Sub-function	03H - Read Update
ES:DI	Buffer Address	Real Mode pointer to the Intel Update structure that will be written with the binary data

Table 11-18. Parameters for the Read Microcode Update Data Function (Contd.)

ECX	Scratch Pad1	Real Mode Segment address of 64 KBytes of RAM Block (lower 16 bits)
ECX	Scratch Pad2	Real Mode Segment address of 64 KBytes of RAM Block (upper 16 bits)
DX	Scratch Pad3	Real Mode Segment address of 64 KBytes of RAM Block
SS:SP	Stack pointer	32 KBytes of Stack Minimum
SI	Update Number	This is the index number of the update block to be read. This value is zero based and must be less than the update count returned from the presence test function.
Output		
CF	Carry Flag	Carry Set - Failure - AH contains Status
Carry Clear - All return values are valid.		
AH	Return Code	Status of the Call
AL	OEM Error	Additional OEM Information
Return Codes (see Table 11-19 for code definitions)		
SUCCESS		The function completed successfully.
READ_FAILURE		There was a failure because of the inability to read the storage device.
UPDATE_NUM_INVALID		Update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY		The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks. The specified block is not a header block and is not empty.

The read function enables the caller to read any microcode update data that already exists in a BIOS and make decisions about the addition of new updates. As a result of a successful call, the BIOS copies the microcode update into the location pointed to by ES:DI, with the contents of all Update block(s) that are used to store the specified microcode update.

If the specified block is not a header block, but does contain valid data from a microcode update that spans multiple update blocks, then the BIOS must return Failure with the NOT_EMPTY error code in AH.

An update block is considered unused and available for storing a new update if its Header Version contains the value 0FFFFFFFFH after return from this function call. The actual implementation of NVRAM storage management is not specified here and is BIOS dependent. As an example, the actual data value used to represent an empty block by the BIOS may be zero, rather than 0FFFFFFFFH. The BIOS is responsible for translating this information into the header provided by this function.

11.11.8.9 Return Codes

After the call has been made, the return codes listed in Table 11-19 are available in the AH register.

Table 11-19. Return Code Definitions

Return Code	Value	Description
SUCCESS	00H	The function completed successfully.
NOT_IMPLEMENTED	86H	The function is not implemented.
ERASE_FAILURE	90H	A failure because of the inability to erase the storage device.
WRITE_FAILURE	91H	A failure because of the inability to write the storage device.
READ_FAILURE	92H	A failure because of the inability to read the storage device.
STORAGE_FULL	93H	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT	94H	The processor stepping does not currently exist in the system.
INVALID_HEADER	95H	The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	96H	The update does not checksum correctly.
SECURITY_FAILURE	97H	The update was rejected by the processor.
INVALID_REVISION	98H	The same or more recent revision of the update exists in the storage device.
UPDATE_NUM_INVALID	99H	The update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY	9AH	The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks. The specified block is not a header block and is not empty.

CHAPTER 12
ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC)

The Advanced Programmable Interrupt Controller (APIC), referred to in the following sections as the local APIC, was introduced into the IA-32 processors with the Pentium processor (see Section 24.27, "Advanced Programmable Interrupt Controller (APIC)") and is included in the P6 family, Pentium 4, Intel Xeon processors, and other more recent Intel 64 and IA-32 processor families (see Section 12.4.2, "Presence of the Local APIC"). The local APIC performs two primary functions for the processor:

- It receives interrupts from the processor's interrupt pins, from internal sources and from an external I/O APIC (or other external interrupt controller). It sends these to the processor core for handling.
- In multiple processor (MP) systems, it sends and receives interprocessor interrupt (IPI) messages to and from other logical processors on the system bus. IPI messages can be used to distribute interrupts among the processors in the system or to execute system wide functions (such as, booting up processors or distributing work among a group of processors).

The external **I/O APIC** is part of Intel's system chipset. Its primary function is to receive external interrupt events from the system and its associated I/O devices and relay them to the local APIC as interrupt messages. In MP systems, the I/O APIC also provides a mechanism for distributing external interrupts to the local APICs of selected processors or groups of processors on the system bus.

This chapter provides a description of the local APIC and its programming interface. It also provides an overview of the interface between the local APIC and the I/O APIC. Contact Intel for detailed information about the I/O APIC.

When a local APIC has sent an interrupt to its processor core for handling, the processor uses the interrupt and exception handling mechanism described in Chapter 7, "Interrupt and Exception Handling." See Section 7.1, "Interrupt and Exception Overview," for an introduction to interrupt and exception handling.

12.1 LOCAL AND I/O APIC OVERVIEW

Each local APIC consists of a set of APIC registers (see Table 12-1) and associated hardware that control the delivery of interrupts to the processor core and the generation of IPI messages. The APIC registers are memory mapped and can be read and written to using the MOV instruction.

Local APICs can receive interrupts from the following sources:

- **Locally connected I/O devices** — These interrupts originate as an edge or level asserted by an I/O device that is connected directly to the processor's local interrupt pins (LINT0 and LINT1). The I/O devices may also be connected to an 8259-type interrupt controller that is in turn connected to the processor through one of the local interrupt pins.
- **Externally connected I/O devices** — These interrupts originate as an edge or level asserted by an I/O device that is connected to the interrupt input pins of an I/O APIC. Interrupts are sent as I/O interrupt messages from the I/O APIC to one or more of the processors in the system.
- **Inter-processor interrupts (IPIs)** — An Intel 64 or IA-32 processor can use the IPI mechanism to interrupt another processor or group of processors on the system bus. IPIs are used for software self-interrupts, interrupt forwarding, or preemptive scheduling.
- **APIC timer generated interrupts** — The local APIC timer can be programmed to send a local interrupt to its associated processor when a programmed count is reached (see Section 12.5.4, "APIC Timer").
- **Performance monitoring counter interrupts** — P6 family, Pentium 4, and Intel Xeon processors provide the ability to send an interrupt to its associated processor when a performance-monitoring counter overflows (see Section 21.6.3.5.8, "Generating an Interrupt on Overflow").
- **Thermal Sensor interrupts** — Pentium 4 and Intel Xeon processors provide the ability to send an interrupt to themselves when the internal thermal sensor has been tripped (see Section 16.8.2, "Thermal Monitor").
- **APIC internal error interrupts** — When an error condition is recognized within the local APIC (such as an attempt to access an unimplemented register), the APIC can be programmed to send an interrupt to its associated processor (see Section 12.5.3, "Error Handling").

Of these interrupt sources: the processor's LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, the thermal sensor, and the internal APIC error detector are referred to as **local interrupt sources**. Upon receiving a signal from a local interrupt source, the local APIC delivers the interrupt to the processor core using an interrupt delivery protocol that has been set up through a group of APIC registers called the **local vector table** or **LVT** (see Section 12.5.1, "Local Vector Table"). A separate entry is provided in the local vector table for each local interrupt source, which allows a specific interrupt delivery protocol to be set up for each source. For example, if the LINT1 pin is going to be used as an NMI pin, the LINT1 entry in the local vector table can be set up to deliver an interrupt with vector number 2 (NMI interrupt) to the processor core.

The local APIC handles interrupts from the other two interrupt sources (externally connected I/O devices and IPIs) through its IPI message handling facilities.

A processor can generate IPIs by programming the interrupt command register (ICR) in its local APIC (see Section 12.6.1, "Interrupt Command Register (ICR)"). The act of writing to the ICR causes an IPI message to be generated and issued on the system bus (for Pentium 4 and Intel Xeon processors) or on the APIC bus (for Pentium and P6 family processors). See Section 12.2, "System Bus Vs. APIC Bus."

IPIs can be sent to other processors in the system or to the originating processor (self-interrupts). When the target processor receives an IPI message, its local APIC handles the message automatically (using information included in the message such as vector number and trigger mode). See Section 12.6, "Issuing Interprocessor Interrupts," for a detailed explanation of the local APIC's IPI message delivery and acceptance mechanism.

The local APIC can also receive interrupts from externally connected devices through the I/O APIC (see Figure 12-1). The I/O APIC is responsible for receiving interrupts generated by system hardware and I/O devices and forwarding them to the local APIC as interrupt messages.

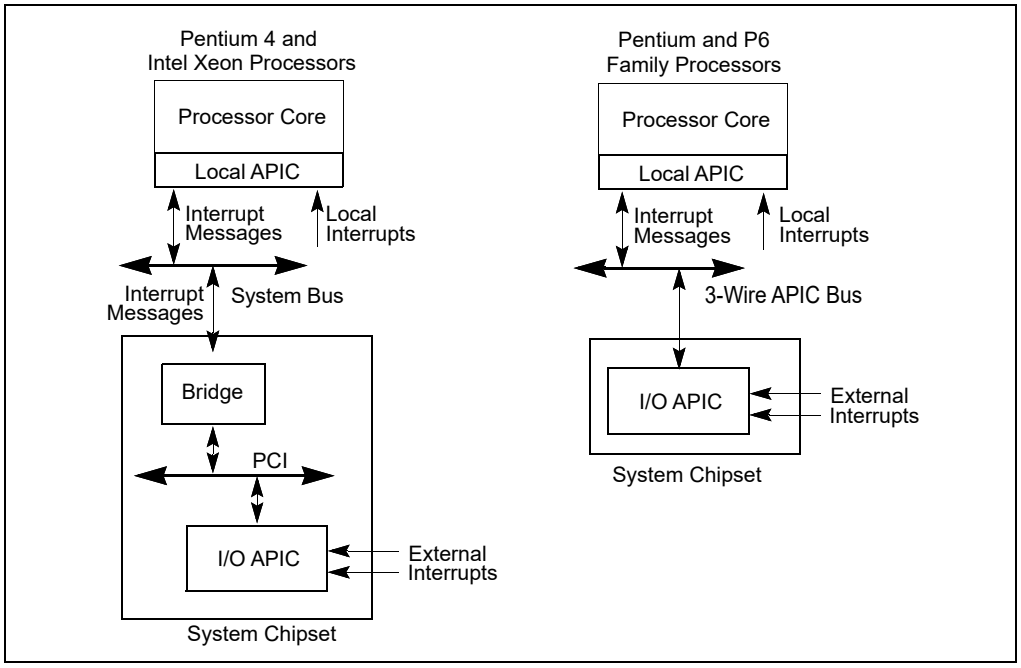


Figure 12-1. Relationship of Local APIC and I/O APIC In Single-Processor Systems

Individual pins on the I/O APIC can be programmed to generate a specific interrupt vector when asserted. The I/O APIC also has a "virtual wire mode" that allows it to communicate with a standard 8259A-style external interrupt controller. Note that the local APIC can be disabled (see Section 12.4.3, "Enabling or Disabling the Local APIC"). This allows an associated processor core to receive interrupts directly from an 8259A interrupt controller.

Both the local APIC and the I/O APIC are designed to operate in MP systems (see Figures 12-2 and 12-3). Each local APIC handles interrupts from the I/O APIC, IPIs from processors on the system bus, and self-generated interrupts. Interrupts can also be delivered to the individual processors through the local interrupt pins; however, this mechanism is commonly not used in MP systems.

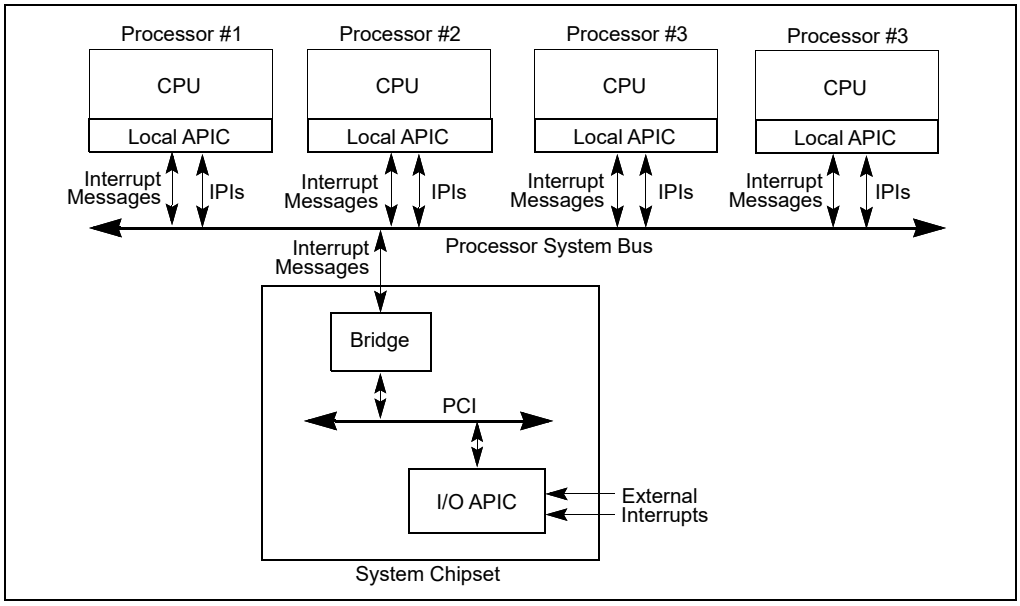


Figure 12-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

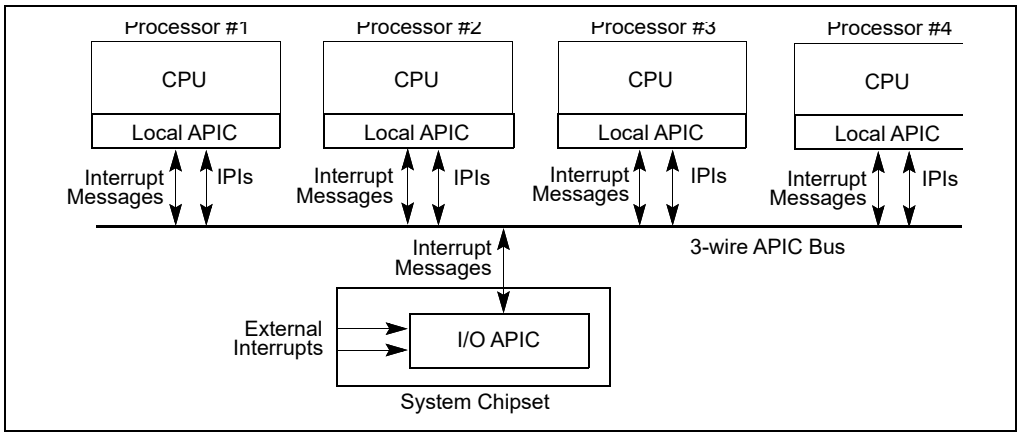


Figure 12-3. Local APICs and I/O APIC When P6 Family Processors Are Used in Multiple-Processor Systems

The IPI mechanism is typically used in MP systems to send fixed interrupts (interrupts for a specific vector number) and special-purpose interrupts to processors on the system bus. For example, a local APIC can use an IPI to forward a fixed interrupt to another processor for servicing. Special-purpose IPIs (including NMI, INIT, SMI, and SIPI IPIs) allow one or more processors on the system bus to perform system-wide boot-up and control functions.

The following sections focus on the local APIC and its implementation in the Pentium 4, Intel Xeon, and P6 family processors. In these sections, the terms "local APIC" and "I/O APIC" refer to local and I/O APICs used with the P6 family processors and to local and I/O xAPICs used with the Pentium 4 and Intel Xeon processors (see Section 12.3, "The Intel® 82489DX External APIC, the APIC, the xAPIC, and the X2APIC").

12.2 SYSTEM BUS VS. APIC BUS

For the P6 family and Pentium processors, the I/O APIC and local APICs communicate through the 3-wire inter-APIC bus (see Figure 12-3). Local APICs also use the APIC bus to send and receive IPIs. The APIC bus and its messages are invisible to software and are not classed as architectural.

Beginning with the Pentium 4 and Intel Xeon processors, the I/O APIC and local APICs (using the xAPIC architecture) communicate through the system bus (see Figure 12-2). The I/O APIC sends interrupt requests to the processors on the system bus through bridge hardware that is part of the Intel chipset. The bridge hardware generates the interrupt messages that go to the local APICs. IPIs between local APICs are transmitted directly on the system bus.

12.3 THE INTEL® 82489DX EXTERNAL APIC, THE APIC, THE XAPIC, AND THE X2APIC

The local APIC in the P6 family and Pentium processors is an architectural subset of the Intel® 82489DX external APIC. See Section 24.27.1, "Software Visible Differences Between the Local APIC and the 82489DX."

The APIC architecture used in the Pentium 4 and Intel Xeon processors (called the xAPIC architecture) is an extension of the APIC architecture found in the P6 family processors. The primary difference between the APIC and xAPIC architectures is that with the xAPIC architecture, the local APICs and the I/O APIC communicate through the system bus. With the APIC architecture, they communication through the APIC bus (see Section 12.2, "System Bus Vs. APIC Bus"). Also, some APIC architectural features have been extended and/or modified in the xAPIC architecture. These extensions and modifications are described in Section 12.4 through Section 12.10.

The basic operating mode of the xAPIC is **xAPIC mode**. The x2APIC architecture is an extension of the xAPIC architecture, primarily to increase processor addressability. The x2APIC architecture provides backward compatibility to the xAPIC architecture and forward extendability for future Intel platform innovations. These extensions and modifications are supported by a new mode of execution (**x2APIC mode**) are detailed in Section 12.12.

12.4 LOCAL APIC

The following sections describe the architecture of the local APIC and how to detect it, identify it, and determine its status. Descriptions of how to program the local APIC are given in Section 12.5.1, "Local Vector Table," and Section 12.6.1, "Interrupt Command Register (ICR)."

12.4.1 The Local APIC Block Diagram

Figure 12-4 gives a functional block diagram for the local APIC. Software interacts with the local APIC by reading and writing its registers. APIC registers are memory-mapped to a 4-KByte region of the processor's physical address space with an initial starting address of FEE00000H. For correct APIC operation, this address space must be mapped to an area of memory that has been designated as strong uncacheable (UC). See Section 13.3, "Methods of Caching Available."

In MP system configurations, the APIC registers for Intel 64 or IA-32 processors on the system bus are initially mapped to the same 4-KByte region of the physical address space. Software has the option of changing initial mapping to a different 4-KByte region for all the local APICs or of mapping the APIC registers for each local APIC to its own 4-KByte region. Section 12.4.5, "Relocating the Local APIC Registers," describes how to relocate the base address for APIC registers.

On processors supporting x2APIC architecture (indicated by CPUID.01H:ECX[21] = 1), the local APIC supports operation both in xAPIC mode and (if enabled by software) in x2APIC mode. x2APIC mode provides extended processor addressability (see Section 12.12).

NOTE

For P6 family, Pentium 4, and Intel Xeon processors, the APIC handles all memory accesses to addresses within the 4-KByte APIC register space internally and no external bus cycles are produced. For the Pentium processors with an on-chip APIC, bus cycles are produced for accesses to the APIC register space. Thus, for software intended to run on Pentium processors, system software should explicitly not map the APIC register space to regular system memory. Doing so can result in an invalid opcode exception (#UD) being generated or unpredictable execution.

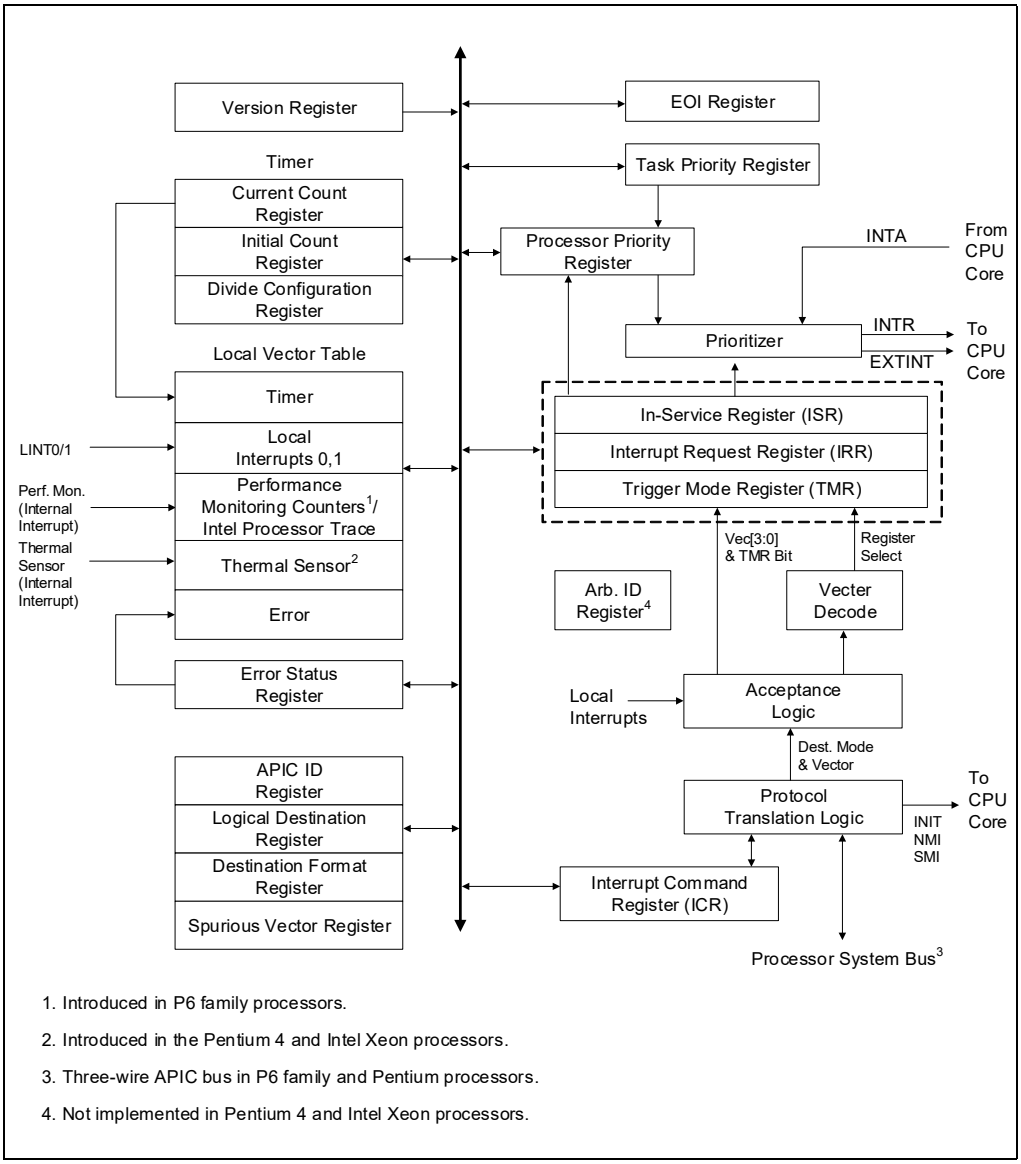


Figure 12-4. Local APIC Structure

Table 12-1 shows how the APIC registers are mapped into the 4-KByte APIC register space. Registers are 32 bits, 64 bits, or 256 bits in width; all are aligned on 128-bit boundaries. All 32-bit registers should be accessed using 128-bit aligned 32-bit loads or stores. Some processors may support loads and stores of less than 32 bits to some of the APIC registers. This is model specific behavior and is not guaranteed to work on all processors. Any FP/MMX/SSE access to an APIC register, or any access that touches bytes 4 through 15 of an APIC register may cause undefined behavior and must not be executed. This undefined behavior could include hangs, incorrect results or unexpected exceptions, including machine checks, and may vary between implementations. Wider registers (64-bit or 256-bit) must be accessed using multiple 32-bit loads or stores, with all accesses being 128-bit aligned.

The local APIC registers listed in Table 12-1 are not MSRs. The only MSR associated with the programming of the local APIC is the IA32_APIC_BASE MSR (see Section 12.4.3, “Enabling or Disabling the Local APIC”).

NOTE

In processors based on Nehalem[1] microarchitecture, the Local APIC ID Register is no longer Read/Write; it is Read Only.

Table 12-1. Local APIC Register Address Map

Address	Register Name	Software Read/Write
FEE0 0000H	Reserved	
FEE0 0010H	Reserved	
FEE0 0020H	Local APIC ID Register	Read/Write.
FEE0 0030H	Local APIC Version Register	Read Only.
FEE0 0040H	Reserved	
FEE0 0050H	Reserved	
FEE0 0060H	Reserved	
FEE0 0070H	Reserved	
FEE0 0080H	Task Priority Register (TPR)	Read/Write.
FEE0 0090H	Arbitration Priority Register[1] (APR)	Read Only.
FEE0 00A0H	Processor Priority Register (PPR)	Read Only.
FEE0 00B0H	EOI Register	Write Only.
FEE0 00C0H	Remote Read Register[1] (RRD)	Read Only
FEE0 00D0H	Logical Destination Register	Read/Write.
FEE0 00E0H	Destination Format Register	Read/Write (see Section 12.6.2.2).
FEE0 00F0H	Spurious Interrupt Vector Register	Read/Write (see Section 12.9.
FEE0 0100H	In-Service Register (ISR); bits 31:0	Read Only.
FEE0 0110H	In-Service Register (ISR); bits 63:32	Read Only.
FEE0 0120H	In-Service Register (ISR); bits 95:64	Read Only.
FEE0 0130H	In-Service Register (ISR); bits 127:96	Read Only.
FEE0 0140H	In-Service Register (ISR); bits 159:128	Read Only.
FEE0 0150H	In-Service Register (ISR); bits 191:160	Read Only.
FEE0 0160H	In-Service Register (ISR); bits 223:192	Read Only.
FEE0 0170H	In-Service Register (ISR); bits 255:224	Read Only.
FEE0 0180H	Trigger Mode Register (TMR); bits 31:0	Read Only.
FEE0 0190H	Trigger Mode Register (TMR); bits 63:32	Read Only.
FEE0 01A0H	Trigger Mode Register (TMR); bits 95:64	Read Only.
FEE0 01B0H	Trigger Mode Register (TMR); bits 127:96	Read Only.
FEE0 01C0H	Trigger Mode Register (TMR); bits 159:128	Read Only.
FEE0 01D0H	Trigger Mode Register (TMR); bits 191:160	Read Only.
FEE0 01E0H	Trigger Mode Register (TMR); bits 223:192	Read Only.
FEE0 01F0H	Trigger Mode Register (TMR); bits 255:224	Read Only.
FEE0 0200H	Interrupt Request Register (IRR); bits 31:0	Read Only.
FEE0 0210H	Interrupt Request Register (IRR); bits 63:32	Read Only.
FEE0 0220H	Interrupt Request Register (IRR); bits 95:64	Read Only.

1. See Table 2-1, "CPUID Signature Values of DisplayFamily_DisplayModel," on page 1, and Section 2.8, "MSRs In Processors Based on Nehalem Microarchitecture," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4, to determine which processors are based on Nehalem microarchitecture.

Table 12-1. Local APIC Register Address Map (Contd.)

Address	Register Name	Software Read/Write
FEE0 0230H	Interrupt Request Register (IRR); bits 127:96	Read Only.
FEE0 0240H	Interrupt Request Register (IRR); bits 159:128	Read Only.
FEE0 0250H	Interrupt Request Register (IRR); bits 191:160	Read Only.
FEE0 0260H	Interrupt Request Register (IRR); bits 223:192	Read Only.
FEE0 0270H	Interrupt Request Register (IRR); bits 255:224	Read Only.
FEE0 0280H	Error Status Register	Write/Read; see Section 12.5.3.
FEE0 0290H through FEE0 02E0H	Reserved	
FEE0 02F0H	LVT Corrected Machine Check Interrupt (CMCI) Register	Read/Write.
FEE0 0300H	Interrupt Command Register (ICR); bits 0-31	Read/Write.
FEE0 0310H	Interrupt Command Register (ICR); bits 32-63	Read/Write.
FEE0 0320H	LVT Timer Register	Read/Write.
FEE0 0330H	LVT Thermal Sensor Register[2]	Read/Write.
FEE0 0340H	LVT Performance Monitoring Counters Register[3]	Read/Write.
FEE0 0350H	LVT LINT0 Register	Read/Write.
FEE0 0360H	LVT LINT1 Register	Read/Write.
FEE0 0370H	LVT Error Register	Read/Write.
FEE0 0380H	Initial Count Register (for Timer)	Read/Write.
FEE0 0390H	Current Count Register (for Timer)	Read Only.
FEE0 03A0H through FEE0 03D0H	Reserved	
FEE0 03E0H	Divide Configuration Register (for Timer)	Read/Write.
FEE0 03F0H	Reserved	

NOTES:

1. Not supported in the Pentium 4 and Intel Xeon processors. The Illegal Register Access bit (7) of the ESR will not be set when writing to these registers.
2. Introduced in the Pentium 4 and Intel Xeon processors. This APIC register and its associated function are implementation dependent and may not be present in future IA-32 or Intel 64 processors.
3. Introduced in the Pentium Pro processor. This APIC register and its associated function are implementation dependent and may not be present in future IA-32 or Intel 64 processors.

12.4.2 Presence of the Local APIC

Beginning with the P6 family processors, the presence or absence of an on-chip local APIC can be detected using the CPUID instruction. When the CPUID instruction is executed with a source operand of 1 in the EAX register, bit 9 of the CPUID feature flags returned in the EDX register indicates the presence (set) or absence (clear) of a local APIC.

12.4.3 Enabling or Disabling the Local APIC

The local APIC can be enabled or disabled in either of two ways:

1. Using the APIC global enable/disable flag in the IA32_APIC_BASE MSR (MSR address 1BH; see Figure 12-5):

— When IA32_APIC_BASE[11] is 0, the processor is functionally equivalent to an IA-32 processor without an on-chip APIC. The CPUID feature flag for the APIC (see Section 12.4.2, "Presence of the Local APIC") is also set to 0.

— When IA32_APIC_BASE[11] is set to 0, processor APICs based on the 3-wire APIC bus cannot be generally re-enabled until a system hardware reset. The 3-wire bus loses track of arbitration that would be necessary for complete re-enabling. Certain APIC functionality can be enabled (for example: performance and thermal monitoring interrupt generation).

— For processors that use Front Side Bus (FSB) delivery of interrupts, software may disable or enable the APIC by setting and resetting IA32_APIC_BASE[11]. A hardware reset is not required to re-start APIC functionality, if software guarantees no interrupt will be sent to the APIC as IA32_APIC_BASE[11] is cleared.

— When IA32_APIC_BASE[11] is set to 0, prior initialization to the APIC may be lost and the APIC may return to the state described in Section 12.4.7.1, "Local APIC State After Power-Up or Reset."

2. Using the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 12-23):

— If IA32_APIC_BASE[11] is 1, software can temporarily disable a local APIC at any time by clearing the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 12-23). The state of the local APIC when in this software-disabled state is described in Section 12.4.7.2, "Local APIC State After It Has Been Software Disabled."

— When the local APIC is in the software-disabled state, it can be re-enabled at any time by setting the APIC software enable/disable flag to 1.

For the Pentium processor, the APICEN pin (which is shared with the PICD1 pin) is used during power-up or reset to disable the local APIC.

Note that each entry in the LVT has a mask bit that can be used to inhibit interrupts from being delivered to the processor from selected local interrupt sources (the LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, Intel® Processor Trace, the thermal sensor, and/or the internal APIC error detector).

12.4.4 Local APIC Status and Location

The status and location of the local APIC are contained in the IA32_APIC_BASE MSR (see Figure 12-5). MSR bit functions are described below:

- **BSP flag, bit 8** — Indicates if the processor is the bootstrap processor (BSP). See Section 10.4, "Multiple-Processor (MP) Initialization." Following a power-up or reset, this flag is set to 1 for the processor selected as the BSP and set to 0 for the remaining processors (APs).
- **APIC Global Enable flag, bit 11** — Enables or disables the local APIC (see Section 12.4.3, "Enabling or Disabling the Local APIC"). This flag is available in the Pentium 4, Intel Xeon, and P6 family processors. It is not guaranteed to be available or available at the same location in future Intel 64 or IA-32 processors.
- **APIC Base field, bits 12 through 35** — Specifies the base address of the APIC registers. This 24-bit value is extended by 12 bits at the low end to form the base address. This automatically aligns the address on a 4-KByte boundary. Following a power-up or reset, the field is set to FEE0 0000H.
- Bits 0 through 7, bits 9 and 10, and bits MAXPHYADDR[1] through 63 in the IA32_APIC_BASE MSR are reserved.

1. The MAXPHYADDR is 36 bits for processors that do not support CPUID leaf 80000008H, or indicated by CPUID.80000008H:EAX[bits 7:0] for processors that support CPUID leaf 80000008H.

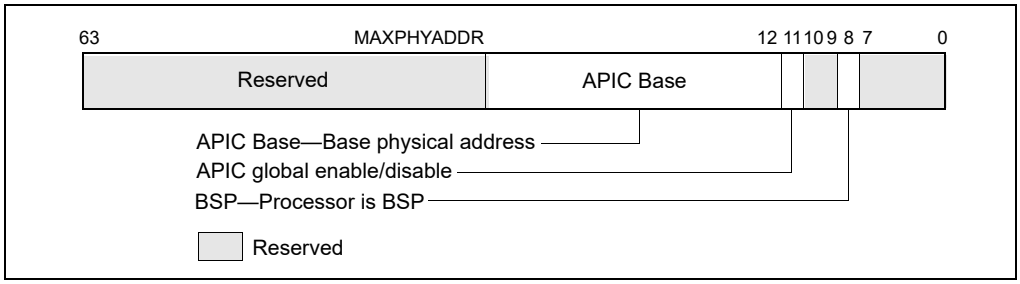


Figure 12-5. IA32_APIC_BASE MSR (APIC_BASE_MSR in P6 Family)

12.4.5 Relocating the Local APIC Registers

The Pentium 4, Intel Xeon, and P6 family processors permit the starting address of the APIC registers to be relocated from FEE00000H to another physical address by modifying the value in the base address field of the IA32_APIC_BASE MSR. This extension of the APIC architecture is provided to help resolve conflicts with memory maps of existing systems and to allow individual processors in an MP system to map their APIC registers to different locations in physical memory.

12.4.6 Local APIC ID

At power up, system hardware assigns a unique APIC ID to each local APIC on the system bus (for Pentium 4 and Intel Xeon processors) or on the APIC bus (for P6 family and Pentium processors). The hardware assigned APIC ID is based on system topology and includes encoding for socket position and cluster information (see Figure 10-2 and Section 10.9.1, "Hierarchical Mapping of Shared Resources").

In MP systems, the local APIC ID is also used as a processor ID by the BIOS and the operating system. Some processors permit software to modify the APIC ID. However, the ability of software to modify the APIC ID is processor model specific. Because of this, operating system software should avoid writing to the local APIC ID register. The value returned by bits 31-24 of the EBX register (when the CPUID instruction is executed with a source operand value of 1 in the EAX register) is always the Initial APIC ID (determined by the platform initialization). This is true even if software has changed the value in the Local APIC ID register.

The processor receives the hardware assigned APIC ID (or Initial APIC ID) by sampling pins A11# and A12# and pins BR0# through BR3# (for the Pentium 4, Intel Xeon, and P6 family processors) and pins BE0# through BE3# (for the Pentium processor). The APIC ID latched from these pins is stored in the APIC ID field of the local APIC ID register (see Figure 12-6), and is used as the Initial APIC ID for the processor.

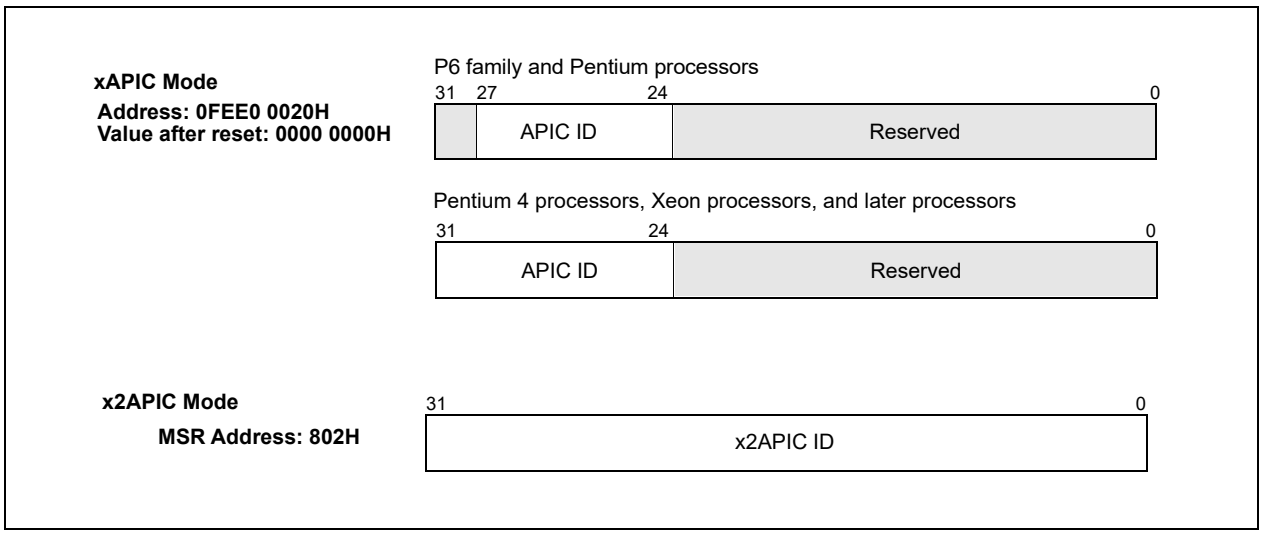


Figure 12-6. Local APIC ID Register

For the P6 family and Pentium processors, the local APIC ID field in the local APIC ID register is 4 bits. Encodings 0H through EH can be used to uniquely identify 15 different processors connected to the APIC bus. For the Pentium 4 and Intel Xeon processors, the xAPIC specification extends the local APIC ID field to 8 bits. These can be used to identify up to 255 processors in the system.

12.4.7 Local APIC State

The following sections describe the state of the local APIC and its registers following a power-up or reset, after the local APIC has been software disabled, following an INIT reset, and following an INIT-deassert message.

x2APIC will introduce 32-bit ID; see Section 12.12.

12.4.7.1 Local APIC State After Power-Up or Reset

Following a power-up or reset of the processor, the state of local APIC and its registers are as follows:

- The following registers are reset to all 0s.
    - IRR, ISR, TMR, ICR, LDR, and TPR.
    - Timer initial count and timer current count registers.
    - Divide configuration register.
- The DFR register is reset to all 1s.
- The LVT register is reset to 0s except for the mask bits; these are set to 1s.
- The local APIC version register is not affected.
- The local APIC ID register is set to a unique APIC ID. (Pentium and P6 family processors only). The Arb ID register is set to the value in the APIC ID register.
- The spurious-interrupt vector register is initialized to 000000FFH. By setting bit 8 to 0, software disables the local APIC.
- If the processor is the only processor in the system or it is the BSP in an MP system (see Section 10.4.1, "BSP and AP Processors"); the local APIC will respond normally to INIT and NMI messages, to INIT# signals and to STPCLK# signals. If the processor is in an MP system and has been designated as an AP; the local APIC will respond the same as for the BSP. In addition, it will respond to SIPI messages. For P6 family processors only, an AP will not respond to a STPCLK# signal.

12.4.7.2 Local APIC State After It Has Been Software Disabled

When the APIC software enable/disable flag in the spurious interrupt vector register has been explicitly cleared (as opposed to being cleared during a power up or reset), the local APIC is temporarily disabled (see Section 12.4.3, "Enabling or Disabling the Local APIC"). The operation and response of a local APIC while in this software-disabled state is as follows:

- The local APIC will respond normally to INIT, NMI, SMI, and SIPI messages.
- Pending interrupts in the IRR and ISR registers are held and require masking or handling by the CPU.
- The local APIC can still issue IPIs. It is software's responsibility to avoid issuing IPIs through the IPI mechanism and the ICR register if sending interrupts through this mechanism is not desired.
- The reception of any interrupt or transmission of any IPIs that are in progress when the local APIC is disabled are completed before the local APIC enters the software-disabled state.
- The mask bits for all the LVT entries are set. Attempts to reset these bits will be ignored.
- (For Pentium and P6 family processors) The local APIC continues to listen to all bus messages in order to keep its arbitration ID synchronized with the rest of the system.

12.4.7.3 Local APIC State After an INIT Reset ("Wait-for-SIPI" State)

An INIT reset of the processor can be initiated in either of two ways:

- By asserting the processor's INIT# pin.
- By sending the processor an INIT IPI (an IPI with the delivery mode set to INIT).

Upon receiving an INIT through either of these mechanisms, the processor responds by beginning the initialization process of the processor core and the local APIC. The state of the local APIC following an INIT reset is the same as it is after a power-up or hardware reset, except that the APIC ID and arbitration ID registers are not affected. This state is also referred to at the "wait-for-SIPI" state (see also: Section 10.4.2, "MP Initialization Protocol Requirements and Restrictions").

12.4.7.4 Local APIC State After It Receives an INIT-Deassert IPI

Only the Pentium and P6 family processors support the INIT-deassert IPI. An INIT-deassert IPI has no affect on the state of the APIC, other than to reload the arbitration ID register with the value in the APIC ID register.

12.4.8 Local APIC Version Register

The local APIC contains a hardwired version register. Software can use this register to identify the APIC version (see Figure 12-7). In addition, the register specifies the number of entries in the local vector table (LVT) for a specific implementation.

The fields in the local APIC version register are as follows:

Version The version numbers of the local APIC:

0XH	82489DX discrete APIC.
10H - 15H	Integrated APIC.

Other values reserved.

Max LVT Entry Shows the number of LVT entries minus 1. For the Pentium 4 and Intel Xeon processors (which have 6 LVT entries), the value returned in the Max LVT field is 5; for the P6 family processors (which have 5 LVT entries), the value returned is 4; for the Pentium processor (which has 4 LVT entries), the value returned is 3. For processors based on the Nehalem microarchitecture (which has 7 LVT entries) and onward, the value returned is 6.

Suppress EOI-broadcasts
Indicates whether software can inhibit the broadcast of EOI message by setting bit 12 of the Spurious Interrupt Vector Register; see Section 12.8.5 and Section 12.9.

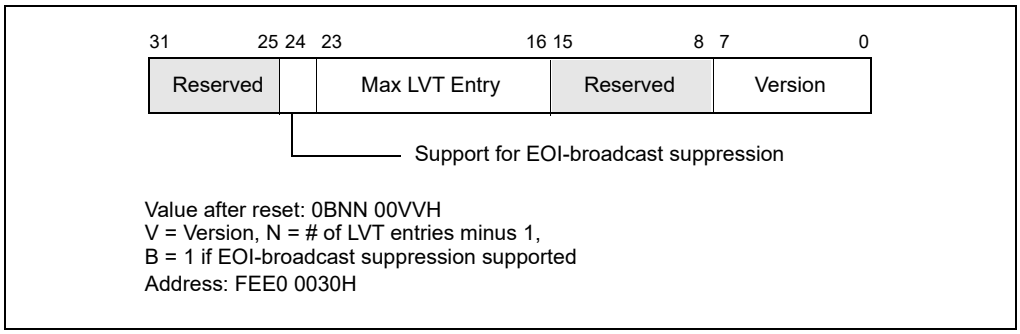


Figure 12-7. Local APIC Version Register

12.5 HANDLING LOCAL INTERRUPTS

The following sections describe facilities that are provided in the local APIC for handling local interrupts. These include: the processor's LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, Intel Processor Trace, the thermal sensor, and the internal APIC error detector. Local interrupt handling facilities include: the LVT, the error status register (ESR), the divide configuration register (DCR), and the initial count and current count registers.

12.5.1 Local Vector Table

The local vector table (LVT) allows software to specify the manner in which the local interrupts are delivered to the processor core. It consists of the following 32-bit APIC registers (see Figure 12-8), one for each local interrupt:

- **LVT CMCI Register (FEE0 02F0H)** — Specifies interrupt delivery when an overflow condition of corrected machine check error count reaching a threshold value occurred in a machine check bank supporting CMCI (see Section 17.5.1, "CMCI Local APIC Interface").
- **LVT Timer Register (FEE0 0320H)** — Specifies interrupt delivery when the APIC timer signals an interrupt (see Section 12.5.4, "APIC Timer").
- **LVT Thermal Monitor Register (FEE0 0330H)** — Specifies interrupt delivery when the thermal sensor generates an interrupt (see Section 16.8.2, "Thermal Monitor"). This LVT entry is implementation specific, not architectural. If implemented, it will always be at base address FEE0 0330H.
- **LVT Performance Counter Register (FEE0 0340H)** — Specifies interrupt delivery when a performance counter generates an interrupt on overflow (see Section 21.6.3.5.8, "Generating an Interrupt on Overflow") or when Intel PT signals a ToPA PMI (see Section 34.2.7.2). This LVT entry is implementation specific, not architectural. If implemented, it is not guaranteed to be at base address FEE0 0340H.
- **LVT LINT0 Register (FEE0 0350H)** — Specifies interrupt delivery when an interrupt is signaled at the LINT0 pin.
- **LVT LINT1 Register (FEE0 0360H)** — Specifies interrupt delivery when an interrupt is signaled at the LINT1 pin.
- **LVT Error Register (FEE0 0370H)** — Specifies interrupt delivery when the APIC detects an internal error (see Section 12.5.3, "Error Handling").

The LVT performance counter register and its associated interrupt were introduced in the P6 processors and are also present in the Pentium 4 and Intel Xeon processors. The LVT thermal monitor register and its associated interrupt were introduced in the Pentium 4 and Intel Xeon processors. The LVT CMCI register and its associated interrupt were introduced in the Intel Xeon 5500 processors.

As shown in Figure 12-8, some of these fields and flags are not available (and reserved) for some entries.

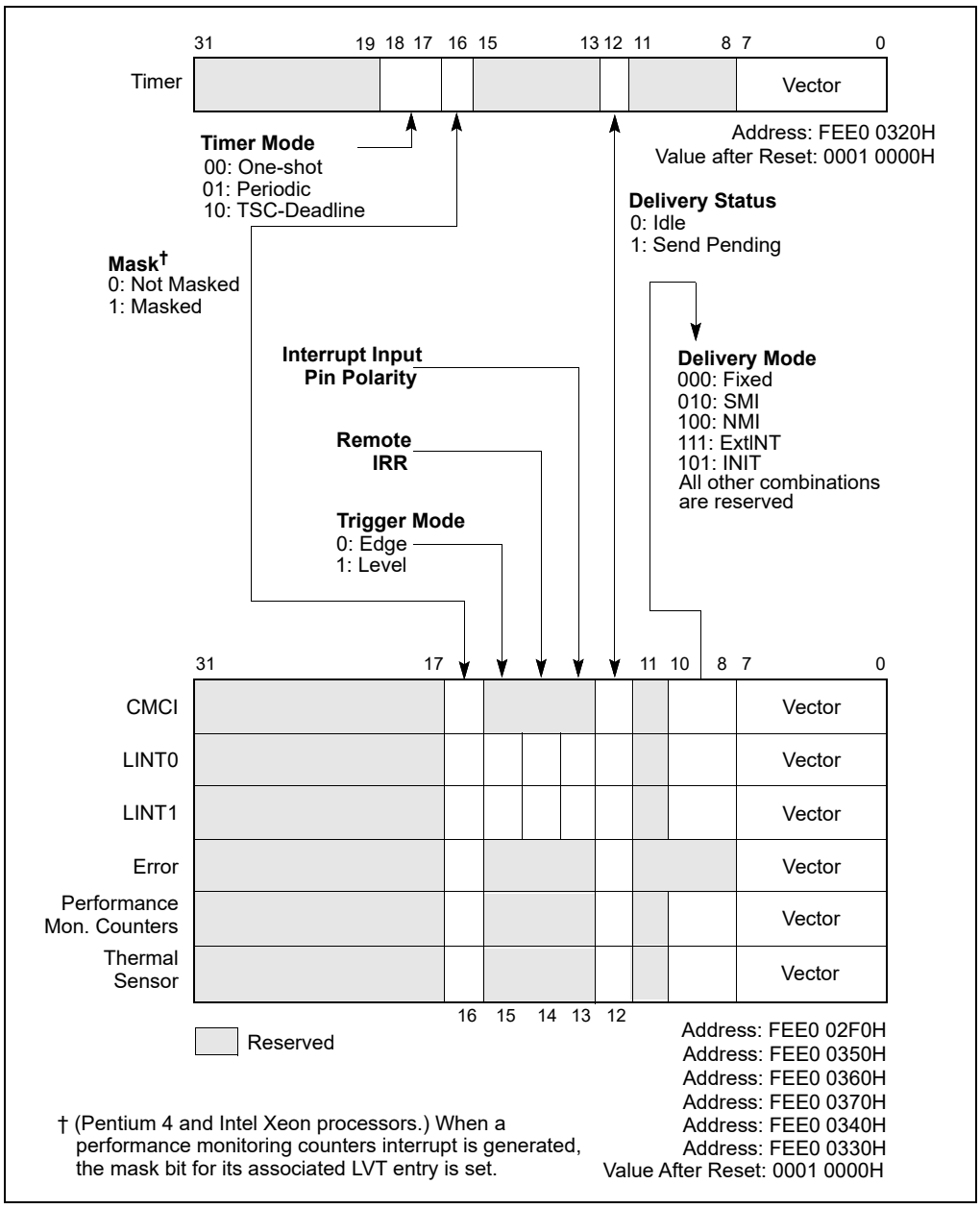


Figure 12-8. Local Vector Table (LVT)

The setup information that can be specified in the registers of the LVT table is as follows:

Vector Interrupt vector number.

Delivery Mode Specifies the type of interrupt to be sent to the processor. Some delivery modes will only operate as intended when used in conjunction with a specific trigger mode. The allowable delivery modes are as follows:

- **000 (Fixed)** Delivers the interrupt specified in the vector field.
- **010 (SMI)** Delivers an SMI interrupt to the processor core through the processor's local SMI signal path. When using this delivery mode, the vector field should be set to 00H for future compatibility.

100 (NMI) Delivers an NMI interrupt to the processor. The vector information is ignored.

101 (INIT) Delivers an INIT request to the processor core, which causes the processor to perform an INIT. When using this delivery mode, the vector field should be set to 00H for future compatibility. Not supported for the LVT CMCI register, the LVT thermal monitor register, or the LVT performance counter register.

110 Reserved; not supported for any LVT register.

111 (ExtINT) Causes the processor to respond to the interrupt as if the interrupt originated in an externally connected (8259A-compatible) interrupt controller. A special INTA bus cycle corresponding to ExtINT, is routed to the external controller. The external controller is expected to supply the vector information. The APIC architecture supports only one ExtINT source in a system, usually contained in the compatibility bridge. Only one processor in the system should have an LVT entry configured to use the ExtINT delivery mode. Not supported for the LVT CMCI register, the LVT thermal monitor register, or the LVT performance counter register.

Delivery Status (Read Only)
Indicates the interrupt delivery status, as follows:

0 (Idle) There is currently no activity for this interrupt source, or the previous interrupt from this source was delivered to the processor core and accepted.

1 (Send Pending)
Indicates that an interrupt from this source has been delivered to the processor core but has not yet been accepted (see Section 12.5.5, "Local Interrupt Acceptance").

Interrupt Input Pin Polarity
Specifies the polarity of the corresponding interrupt pin: (0) active high or (1) active low.

Remote IRR Flag (Read Only)
For fixed mode, level-triggered interrupts; this flag is set when the local APIC accepts the interrupt for servicing and is reset when an EOI command is received from the processor. The meaning of this flag is undefined for edge-triggered interrupts and other delivery modes.

Trigger Mode Selects the trigger mode for the local LINT0 and LINT1 pins: (0) edge sensitive and (1) level sensitive. This flag is only used when the delivery mode is Fixed. When the delivery mode is NMI, SMI, or INIT, the trigger mode is always edge sensitive. When the delivery mode is ExtINT, the trigger mode is always level sensitive. The timer and error interrupts are always treated as edge sensitive.

If the local APIC is not used in conjunction with an I/O APIC and fixed delivery mode is selected; the Pentium 4, Intel Xeon, and P6 family processors will always use level-sensitive triggering, regardless if edge-sensitive triggering is selected.

Software should always set the trigger mode in the LVT LINT1 register to 0 (edge sensitive). Level-sensitive interrupts are not supported for LINT1.

Mask Interrupt mask: (0) enables reception of the interrupt and (1) inhibits reception of the interrupt. When the local APIC handles a performance-monitoring counters interrupt, it automatically sets the mask flag in the LVT performance counter register. This flag is set to 1 on reset. It can be cleared only by software.

Timer Mode Bits 18:17 selects the timer mode (see Section 12.5.4):

(00b) one-shot mode using a count-down value,

(01b) periodic mode reloading a count-down value,

(10b) TSC-Deadline mode using absolute target value in IA32_TSC_DEADLINE MSR (see Section 12.5.4.1),

(11b) is reserved.

12.5.2 Valid Interrupt Vectors

The Intel 64 and IA-32 architectures define 256 vector numbers, ranging from 0 through 255 (see Section 7.2, "Exception and Interrupt Vectors"). Local and I/O APICs support 240 of these vectors (in the range of 16 to 255) as valid interrupts.

When an interrupt vector in the range of 0 to 15 is sent or received through the local APIC, the APIC indicates an illegal vector in its Error Status Register (see Section 12.5.3, "Error Handling"). The Intel 64 and IA-32 architectures reserve vectors 16 through 31 for predefined interrupts, exceptions, and Intel-reserved encodings (see Table 7-1). However, the local APIC does not treat vectors in this range as illegal.

When an illegal vector value (0 to 15) is written to an LVT entry and the delivery mode is Fixed (bits 8-11 equal 0), the APIC may signal an illegal vector error, without regard to whether the mask bit is set or whether an interrupt is actually seen on the input.

12.5.3 Error Handling

The local APIC records errors detected during interrupt handling in the error status register (ESR). The format of the ESR is given in Figure 12-9; it contains the following flags:

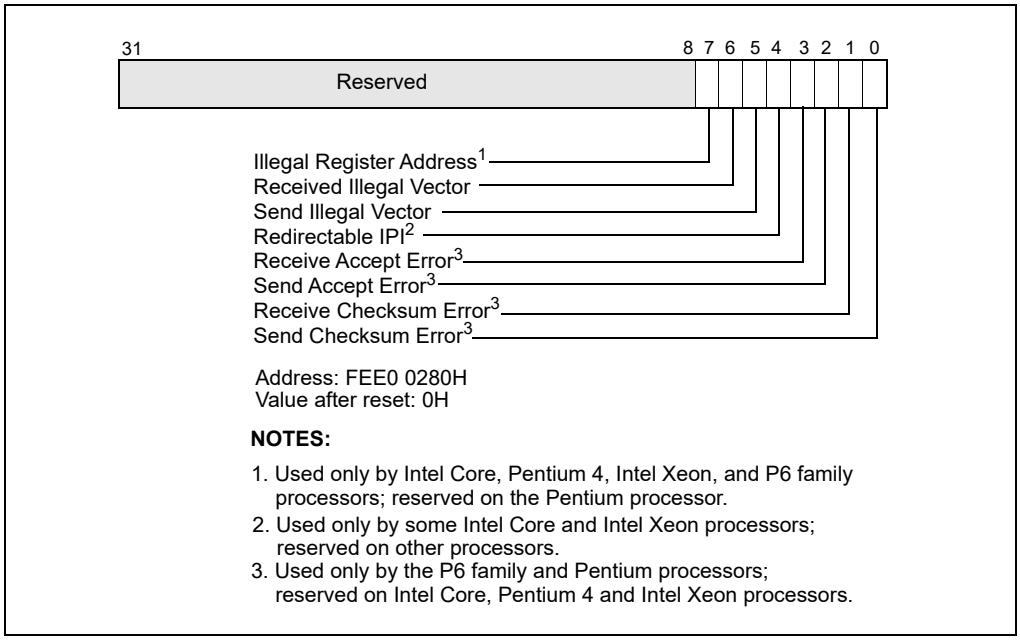


Figure 12-9. Error Status Register (ESR)

- Bit 0: Send Checksum Error.
  Set when the local APIC detects a checksum error for a message that it sent on the APIC bus. Used only on P6 family and Pentium processors.
- Bit 1: Receive Checksum Error.
  Set when the local APIC detects a checksum error for a message that it received on the APIC bus. Used only on P6 family and Pentium processors.
- Bit 2: Send Accept Error.
  Set when the local APIC detects that a message it sent was not accepted by any APIC on the APIC bus. Used only on P6 family and Pentium processors.
- Bit 3: Receive Accept Error.
  Set when the local APIC detects that the message it received was not accepted by any APIC on the APIC bus, including itself. Used only on P6 family and Pentium processors.

- Bit 4: Redirectable IPI.
  Set when the local APIC detects an attempt to send an IPI with the lowest-priority delivery mode and the local APIC does not support the sending of such IPIs. This bit is used on some Intel Core and Intel Xeon processors. As noted in Section 12.6.2, the ability of a processor to send a lowest-priority IPI is model-specific and should be avoided.
- Bit 5: Send Illegal Vector.
  Set when the local APIC detects an illegal vector (one in the range 0 to 15) in the message that it is sending. This occurs as the result of a write to the ICR (in both xAPIC and x2APIC modes) or to SELF IPI register (x2APIC mode only) with an illegal vector.

  If the local APIC does not support the sending of lowest-priority IPIs and software writes the ICR to send a lowest-priority IPI with an illegal vector, the local APIC sets only the "redirectable IPI" error bit. The interrupt is not processed and hence the "Send Illegal Vector" bit is not set in the ESR.
- Bit 6: Receive Illegal Vector.
  Set when the local APIC detects an illegal vector (one in the range 0 to 15) in an interrupt message it receives or in an interrupt generated locally from the local vector table or via a self IPI. Such interrupts are not delivered to the processor; the local APIC will never set an IRR bit in the range 0 to 15.
- Bit 7: Illegal Register Address
  Set when the local APIC is in xAPIC mode and software attempts to access a register that is reserved in the processor's local-APIC register-address space; see Table 10-1. (The local-APIC register-address space comprises the 4 KBytes at the physical address specified in the IA32_APIC_BASE MSR.) Used only on Intel Core, Intel Atom, Pentium 4, Intel Xeon, and P6 family processors.

  In x2APIC mode, software accesses the APIC registers using the RDMSR and WRMSR instructions. Use of one of these instructions to access a reserved register cause a general-protection exception (see Section 10.12.1.3). They do not set the "Illegal Register Access" bit in the ESR.

The ESR is a write/read register. Before attempt to read from the ESR, software should first write to it. (The value written does not affect the values read subsequently; only zero may be written in x2APIC mode.) This write clears any previously logged errors and updates the ESR with any errors detected since the last write to the ESR. This write also rearms the APIC error interrupt triggering mechanism.

The LVT Error Register (see Section 12.5.1) allows specification of the vector of the interrupt to be delivered to the processor core when APIC error is detected. The register also provides a means of masking an APIC-error interrupt. This masking only prevents delivery of APIC-error interrupts; the APIC continues to record errors in the ESR.

12.5.4 APIC Timer

The local APIC unit contains a 32-bit programmable timer that is available to software to time events or operations. This timer is set up by programming four registers: the divide configuration register (see Figure 12-10), the initial-count and current-count registers (see Figure 12-11), and the LVT timer register (see Figure 12-8).

If CPUID.06H:EAX.ARAT[bit 2] = 1, the processor's APIC timer runs at a constant rate regardless of P-state transitions and it continues to run at the same rate in deep C-states.

If CPUID.06H:EAX.ARAT[bit 2] = 0 or if CPUID 06H is not supported, the APIC timer may temporarily stop while the processor is in deep C-states or during transitions caused by Enhanced Intel SpeedStep® Technology.

The APIC timer frequency will be the processor's bus clock or core crystal clock frequency (when TSC/core crystal clock ratio is enumerated in CPUID leaf 0x15) divided by the value specified in the divide configuration register.

The timer can be configured through the timer LVT entry for one-shot or periodic operation. In one-shot mode, the timer is started by programming its initial-count register. The initial count value is then copied into the current-count register and count-down begins. After the timer reaches zero, a timer interrupt is generated and the timer remains at its 0 value until reprogrammed.

In periodic mode, the timer is started by writing to the initial-count register (as in one-shot mode), and the value written is copied into the current-count register, which counts down. The current-count register is automatically reloaded from the initial-count register when the count reaches 0 and a timer interrupt is generated, and the count-down is repeated. If during the count-down process the initial-count register is set, counting will restart, using the new initial-count value. The initial-count register is a read-write register; the current-count register is read only.

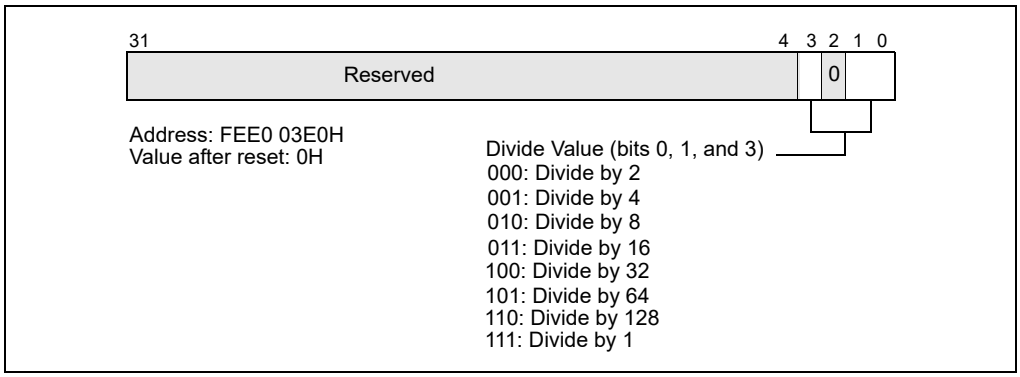


Figure 12-10. Divide Configuration Register

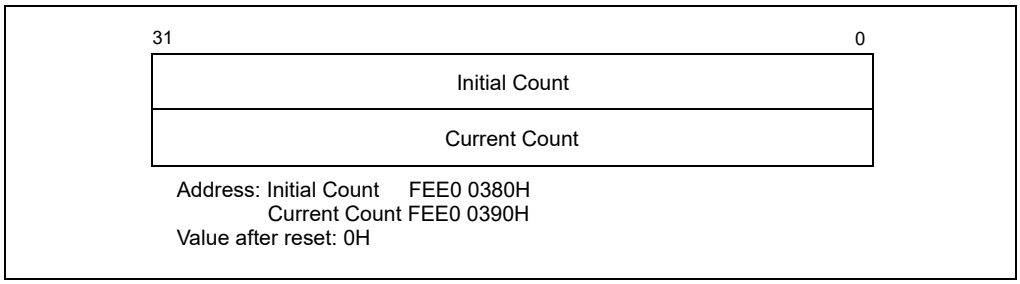


Figure 12-11. Initial Count and Current Count Registers

A write of 0 to the initial-count register effectively stops the local APIC timer, in both one-shot and periodic mode.

The LVT timer register determines the vector number that is delivered to the processor with the timer interrupt that is generated when the timer count reaches zero. The mask flag in the LVT timer register can be used to mask the timer interrupt.

NOTE

Changing the mode of the APIC timer (from one-shot to periodic or vice versa) by writing to the timer LVT entry does not start the timer. To start the timer, it is necessary to write to the initial-count register as described above.

12.5.4.1 TSC-Deadline Mode

The mode of operation of the local-APIC timer is determined by the LVT Timer Register. Specifically:

- If CPUID.01H:ECX.TSC_Deadline[bit 24] = 0, the mode is determined by bit 17 of the register.
- If CPUID.01H:ECX.TSC_Deadline[bit 24] = 1, the mode is determined by bits 18:17. See Figure 12-8. (If CPUID.01H:ECX.TSC_Deadline[bit 24] = 0, bit 18 of the register is reserved.)

The supported timer modes are given in Table 12-2. The three modes of the local APIC timer are mutually exclusive.

Table 12-2. Local APIC Timer Modes

LVT Bits [18:17]	Timer Mode
00b	One-shot mode, program count-down value in an initial-count register. See Section 12.5.4
01b	Periodic mode, program interval value in an initial-count register. See Section 12.5.4
10b	TSC-Deadline mode, program target value in IA32_TSC_DEADLINE MSR.
11b	Reserved

TSC-deadline mode allows software to use the local APIC timer to signal an interrupt at an absolute time. In TSC-deadline mode, writes to the initial-count register are ignored; and current-count register always reads 0. Instead, timer behavior is controlled using the IA32_TSC_DEADLINE MSR.

The IA32_TSC_DEADLINE MSR (MSR address 6E0H) is a per-logical processor MSR that specifies the time at which a timer interrupt should occur. Writing a non-zero 64-bit value into IA32_TSC_DEADLINE arms the timer. An interrupt is generated when the logical processor's time-stamp counter equals or exceeds the target value in the IA32_TSC_DEADLINE MSR.[1] When the timer generates an interrupt, it disarms itself and clears the IA32_TSC_DEADLINE MSR. Thus, each write to the IA32_TSC_DEADLINE MSR generates at most one timer interrupt.

In TSC-deadline mode, writing 0 to the IA32_TSC_DEADLINE MSR disarms the local-APIC timer. Transitioning between TSC-deadline mode and other timer modes also disarms the timer.

The hardware reset value of the IA32_TSC_DEADLINE MSR is 0. In other timer modes (LVT bit 18 = 0), the IA32_TSC_DEADLINE MSR reads zero and writes are ignored.

Software can configure the TSC-deadline timer to deliver a single interrupt using the following algorithm:

1. Detect support for TSC-deadline mode by verifying CPUID.1:ECX.24 = 1.
2. Select the TSC-deadline mode by programming bits 18:17 of the LVT Timer register with 10b.
3. Program the IA32_TSC_DEADLINE MSR with the target TSC value at which the timer interrupt is desired. This causes the processor to arm the timer.
4. The processor generates a timer interrupt when the value of time-stamp counter is greater than or equal to that of IA32_TSC_DEADLINE. It then disarms the timer and clear the IA32_TSC_DEADLINE MSR. (Both the time-stamp counter and the IA32_TSC_DEADLINE MSR are 64-bit unsigned integers.)
5. Software can re-arm the timer by repeating step 3.

The following are usage guidelines for TSC-deadline mode:

- Writes to the IA32_TSC_DEADLINE MSR are not serialized. Therefore, system software should not use WRMSR to the IA32_TSC_DEADLINE MSR as a serializing instruction. Read and write accesses to the IA32_TSC_DEADLINE and other MSR registers will occur in program order.
- Software can disarm the timer at any time by writing 0 to the IA32_TSC_DEADLINE MSR.
- If timer is armed, software can change the deadline (forward or backward) by writing a new value to the IA32_TSC_DEADLINE MSR.
- If software disarms the timer or postpones the deadline, race conditions may result in delivery of a timer interrupt associated with the original deadline. If the deadline has been postponed, software can identify such interrupts by reading the time-stamp counter and comparing its value to the new deadline.[1]
- In xAPIC mode (in which the local-APIC registers are memory-mapped), software must order the memory-mapped write to the LVT entry that enables TSC-deadline mode and any subsequent WRMSR to the IA32_TSC_DEADLINE MSR. Software can assure proper ordering by executing the MFENCE instruction after the memory-mapped write and before any WRMSR. (In x2APIC mode, the WRMSR instruction is used to write to the LVT entry. The processor ensures the ordering of this write and any subsequent WRMSR to the deadline; no fencing is required.)

12.5.5 Local Interrupt Acceptance

When a local interrupt is sent to the processor core, it is subject to the acceptance criteria specified in the interrupt acceptance flow chart in Figure 12-17. If the interrupt is accepted, it is logged into the IRR register and handled by the processor according to its priority (see Section 12.8.4, "Interrupt Acceptance for Fixed Interrupts"). If the interrupt is not accepted, it is sent back to the local APIC and retried.

---

1. If the logical processor is in VMX non-root operation, a read of the time-stamp counter (using either RDMSR, RDTSC, or RDTSCP) may not return the actual value of the time-stamp counter; see Chapter 27 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C. It is the responsibility of software operating in VMX root operation to coordinate the virtualization of the time-stamp counter and the IA32_TSC_DEADLINE MSR.

12.6 ISSUING INTERPROCESSOR INTERRUPTS

The following sections describe the local APIC facilities that are provided for issuing interprocessor interrupts (IPIs) from software. The primary local APIC facility for issuing IPIs is the interrupt command register (ICR). The ICR can be used for the following functions:

- To send an interrupt to another processor.
- To allow a processor to forward an interrupt that it received but did not service to another processor for servicing.
- To direct the processor to interrupt itself (perform a self interrupt).
- To deliver special IPIs, such as the start-up IPI (SIPI) message, to other processors.

Interrupts generated with this facility are delivered to the other processors in the system through the system bus (for Pentium 4 and Intel Xeon processors) or the APIC bus (for P6 family and Pentium processors). The ability for a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.

12.6.1 Interrupt Command Register (ICR)

The interrupt command register (ICR) is a 64-bit[1] local APIC register (see Figure 12-12) that allows software running on the processor to specify and send interprocessor interrupts (IPIs) to other processors in the system.

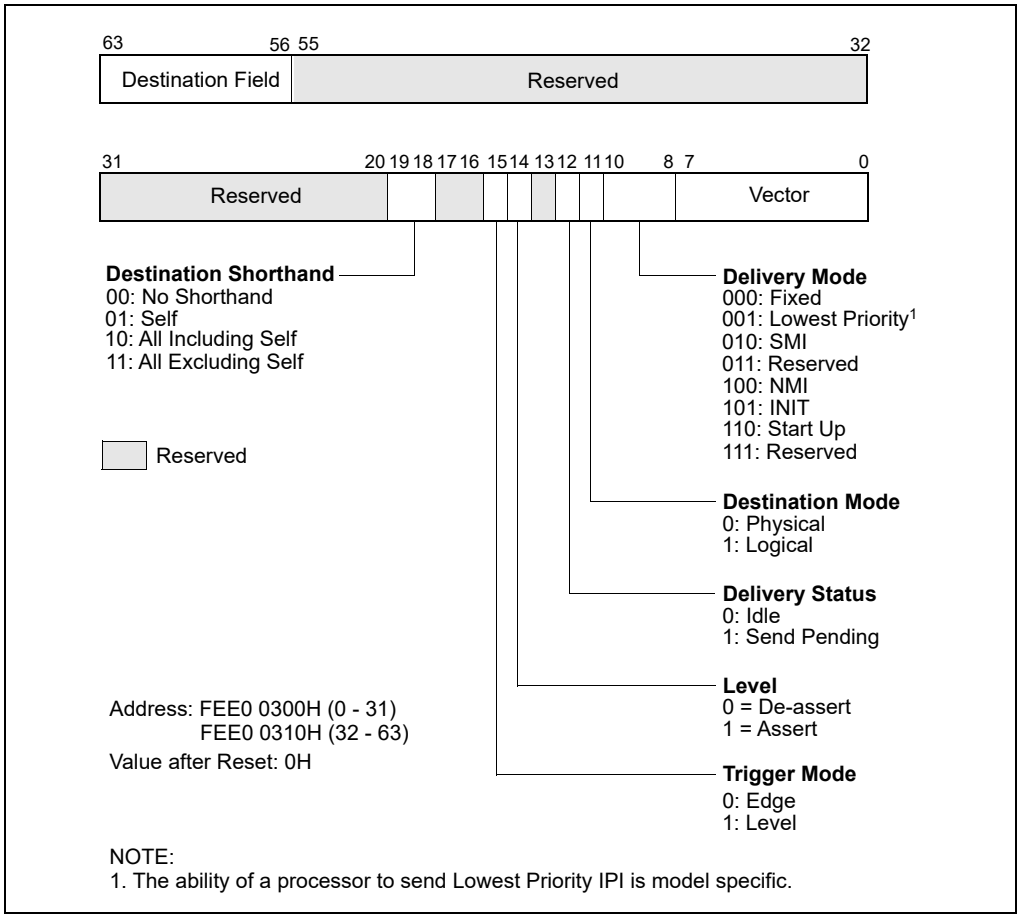


Figure 12-12. Interrupt Command Register (ICR)

1. In XAPIC mode the ICR is addressed as two 32-bit registers, ICR_LOW (FFE0 0300H) and ICR_HIGH (FFE0 0310H). In x2APIC mode, the ICR uses MSR 830H.

To send an IPI, software must set up the ICR to indicate the type of IPI message to be sent and the destination processor or processors. (All fields of the ICR are read-write by software with the exception of the delivery status field, which is read-only.) The act of writing to the low doubleword of the ICR causes the IPI to be sent.

The ICR consists of the following fields.

Vector The vector number of the interrupt being sent.

Delivery Mode Specifies the type of IPI to be sent. This field is also know as the IPI message type field.

- **000 (Fixed)** Delivers the interrupt specified in the vector field to the target processor or processors.
- **001 (Lowest Priority)** Same as fixed mode, except that the interrupt is delivered to the processor executing at the lowest priority among the set of processors specified in the destination field. The ability for a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.
- **010 (SMI)** Delivers an SMI interrupt to the target processor or processors. The vector field must be programmed to 00H for future compatibility.
- **011 (Reserved)**
- **100 (NMI)** Delivers an NMI interrupt to the target processor or processors. The vector information is ignored.
- **101 (INIT)** Delivers an INIT request to the target processor or processors, which causes them to perform an INIT. As a result of this IPI message, all the target processors perform an INIT. The vector field must be programmed to 00H for future compatibility.
- **101 (INIT Level De-assert)** (Not supported in the Pentium 4 and Intel Xeon processors.) Sends a synchronization message to all the local APICs in the system to set their arbitration IDs (stored in their Arb ID registers) to the values of their APIC IDs (see Section 12.7, "System and APIC Bus Arbitration"). For this delivery mode, the level flag must be set to 0 and trigger mode flag to 1. This IPI is sent to all processors, regardless of the value in the destination field or the destination shorthand field; however, software should specify the "all including self" shorthand.
- **110 (Start-Up)** Sends a special "start-up" IPI (called a SIPI) to the target processor or processors. The vector typically points to a start-up routine that is part of the BIOS boot-strap code (see Section 10.4, "Multiple-Processor (MP) Initialization"). IPIs sent with this delivery mode are not automatically retried if the source APIC is unable to deliver it. It is up to the software to determine if the SIPI was not successfully delivered and to reissue the SIPI if necessary.

Destination Mode Selects either physical (0) or logical (1) destination mode (see Section 12.6.2, "Determining IPI Destination").

Delivery Status (Read Only) Indicates the IPI delivery status, as follows:

- **0 (Idle)** Indicates that this local APIC has completed sending any previous IPIs.
- **1 (Send Pending)** Indicates that this local APIC has not completed sending the last IPI.

Level For the INIT level de-assert delivery mode this flag must be set to 0; for all other delivery modes it must be set to 1. (This flag has no meaning in Pentium 4 and Intel Xeon processors, and will always be issued as a 1.)

Trigger Mode Selects the trigger mode when using the INIT level de-assert delivery mode: edge (0) or level (1). It is ignored for all other delivery modes. (This flag has no meaning in Pentium 4 and Intel Xeon processors, and will always be issued as a 0.)

Destination Shorthand

Indicates whether a shorthand notation is used to specify the destination of the interrupt and, if so, which shorthand is used. Destination shorthands are used in place of the 8-bit destination field, and can be sent by software using a single write to the low doubleword of the ICR. Shorthands are defined for the following cases: software self interrupt, IPIs to all processors in the system including the sender, IPIs to all processors in the system excluding the sender.

00: (No Shorthand)
The destination is specified in the destination field.

01: (Self) The issuing APIC is the one and only destination of the IPI. This destination shorthand allows software to interrupt the processor on which it is executing. An APIC implementation is free to deliver the self-interrupt message internally or to issue the message to the bus and "snoop" it as with any other IPI message.

10: (All Including Self)
The IPI is sent to all processors in the system including the processor sending the IPI. The APIC will broadcast an IPI message with the destination field set to FH for Pentium and P6 family processors and to FFH for Pentium 4 and Intel Xeon processors.

11: (All Excluding Self)
The IPI is sent to all processors in a system with the exception of the processor sending the IPI. The APIC broadcasts a message with the physical destination mode and destination field set to FH for Pentium and P6 family processors and to FFH for Pentium 4 and Intel Xeon processors. Support for this destination shorthand in conjunction with the lowest-priority delivery mode is model specific. For Pentium 4 and Intel Xeon processors, when this shorthand is used together with lowest priority delivery mode, the IPI may be redirected back to the issuing processor.

Destination Specifies the target processor or processors. This field is only used when the destination shorthand field is set to 00B. If the destination mode is set to physical, then bits 56 through 59 contain the APIC ID of the target processor for Pentium and P6 family processors and bits 56 through 63 contain the APIC ID of the target processor the for Pentium 4 and Intel Xeon processors. If the destination mode is set to logical, the interpretation of the 8-bit destination field depends on the settings of the DFR and LDR registers of the local APICs in all the processors in the system (see Section 12.6.2, "Determining IPI Destination").

Not all combinations of options for the ICR are valid. Table 12-3 shows the valid combinations for the fields in the ICR for the Pentium 4 and Intel Xeon processors; Table 12-4 shows the valid combinations for the fields in the ICR for the P6 family processors. Also note that the lower half of the ICR may not be preserved over transitions to the deepest C-States.

ICR operation in x2APIC mode is discussed in Section 12.12.9.

Table 12-3. Valid Combinations for Pentium 4 and Intel Xeon Processors Local xAPIC Interrupt Command Register

Destination Shorthand	Valid/Invalid	Trigger Mode	Delivery Mode	Destination Mode
No Shorthand	Valid	Edge	All Modes[1]	Physical or Logical
No Shorthand	Invalid[2]	Level	All Modes	Physical or Logical
Self	Valid	Edge	Fixed	X[3]
Self	Invalid[2]	Level	Fixed	X
Self	Invalid	X	Lowest Priority, NMI, INIT, SMI, Start-Up	X
All Including Self	Valid	Edge	Fixed	X
All Including Self	Invalid[2]	Level	Fixed	X

Table 12-3. Valid Combinations for Pentium 4 and Intel Xeon Processors Local xAPIC Interrupt Command Register

Destination Shorthand	Valid/Invalid	Trigger Mode	Delivery Mode	Destination Mode
All Including Self	Invalid	X	Lowest Priority, NMI, INIT, SMI, Start-Up	X
All Excluding Self	Valid	Edge	Fixed, Lowest Priority[1, 4], NMI, INIT, SMI, Start-Up	X
All Excluding Self	Invalid[2]	Level	FIxed, Lowest Priority[4], NMI, INIT, SMI, Start-Up	X

NOTES:

1. The ability of a processor to send a lowest priority IPI is model specific.
2. For these interrupts, if the trigger mode bit is 1 (Level), the local xAPIC will override the bit setting and issue the interrupt as an edge triggered interrupt.
3. X means the setting is ignored.
4. When using the "lowest priority" delivery mode and the "all excluding self" destination, the IPI can be redirected back to the issuing APIC, which is essentially the same as the "all including self" destination mode.

Table 12-4. Valid Combinations for the P6 Family Processor Local APIC Interrupt Command Register

Destination Shorthand	Valid/Invalid	Trigger Mode	Delivery Mode	Destination Mode
No Shorthand	Valid	Edge	All Modes[1]	Physical or Logical
No Shorthand	Valid[2]	Level	Fixed, Lowest Priority[1], NMI	Physical or Logical
No Shorthand	Valid[3]	Level	INIT	Physical or Logical
Self	Valid	Edge	Fixed	X[4]
Self	Valid[2]	Level	Fixed	X
Self	Invalid[5]	X	Lowest Priority, NMI, INIT, SMI, Start-Up	X
All including Self	Valid	Edge	Fixed	X
All including Self	Valid[2]	Level	Fixed	X
All including Self	Invalid[5]	X	Lowest Priority, NMI, INIT, SMI, Start-Up	X
All excluding Self	Valid	Edge	All Modes[1]	X
All excluding Self	Valid[2]	Level	Fixed, Lowest Priority[1], NMI	X
All excluding Self	Invalid[5]	Level	SMI, Start-Up	X
All excluding Self	Valid[3]	Level	INIT	X
X	Invalid[5]	Level	SMI, Start-Up	X

NOTES:

1. The ability of a processor to send a lowest priority IPI is model specific.
2. Treated as edge triggered if level bit is set to 1, otherwise ignored.
3. Treated as edge triggered when Level bit is set to 1; treated as "INIT Level Deassert" message when level bit is set to 0 (deassert). Only INIT level deassert messages are allowed to have the level bit set to 0. For all other messages the level bit must be set to 1.
4. X means the setting is ignored.
5. The behavior of the APIC is undefined.

12.6.2 Determining IPI Destination

The destination of an IPI[1] can be one, all, or a subset (group) of the processors on the system bus. The sender of the IPI specifies the destination of an IPI with the following APIC registers and fields within the registers:

- **ICR Register** — The following fields in the ICR register are used to specify the destination of an IPI.

1. Determination of IPI destinations in x2APIC mode is discussed in Section 10.12.10.

  - **Destination Mode** — Selects one of two destination modes (physical or logical).
  - **Destination Field** — In physical destination mode, used to specify the APIC ID of the destination processor; in logical destination mode, used to specify a message destination address (MDA) that can be used to select specific processors in clusters.
  - **Destination Shorthand** — A quick method of specifying all processors, all excluding self, or self as the destination.
  - **Delivery mode, Lowest Priority** — Architecturally specifies that a lowest-priority arbitration mechanism be used to select a destination processor from a specified group of processors. The ability of a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.
- **Local destination register (LDR)** — Used in conjunction with the logical destination mode and MDAs to select the destination processors.
- **Destination format register (DFR)** — Used in conjunction with the logical destination mode and MDAs to select the destination processors.

How the ICR, LDR, and DFR are used to select an IPI destination depends on the destination mode used: physical, logical, broadcast/self, or lowest-priority delivery mode. These destination modes are described in the following sections.

12.6.2.1 Physical Destination Mode

In physical destination mode, the destination processor is specified by its local APIC ID (see Section 12.4.6, "Local APIC ID"). For Pentium 4 and Intel Xeon processors, either a single destination (local APIC IDs 00H through FEH) or a broadcast to all APICs (the APIC ID is FFH) may be specified in physical destination mode.

A broadcast IPI (bits 28-31 of the MDA are 1's) or I/O subsystem initiated interrupt with lowest priority delivery mode is not supported in physical destination mode and must not be configured by software. Also, for any non-broadcast IPI or I/O subsystem initiated interrupt with lowest priority delivery mode, software must ensure that APICs defined in the interrupt address are present and enabled to receive interrupts.

For the P6 family and Pentium processors, a single destination is specified in physical destination mode with a local APIC ID of 0H through 0EH, allowing up to 15 local APICs to be addressed on the APIC bus. A broadcast to all local APICs is specified with 0FH.

NOTE

The number of local APICs that can be addressed on the system bus may be restricted by hardware.

12.6.2.2 Logical Destination Mode

In logical destination mode, IPI destination is specified using an 8-bit message destination address (MDA), which is entered in the destination field of the ICR. Upon receiving an IPI message that was sent using logical destination mode, a local APIC compares the MDA in the message with the values in its LDR and DFR to determine if it should accept and handle the IPI. For both configurations of logical destination mode, when combined with lowest priority delivery mode, software is responsible for ensuring that all of the local APICs included in or addressed by the IPI or I/O subsystem interrupt are present and enabled to receive the interrupt.

Figure 12-13 shows the layout of the logical destination register (LDR). The 8-bit logical APIC ID field in this register is used to create an identifier that can be compared with the MDA.

NOTE

The logical APIC ID should not be confused with the local APIC ID that is contained in the local APIC ID register.

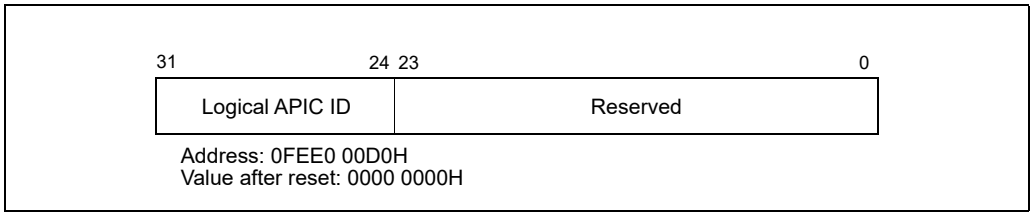


Figure 12-13. Logical Destination Register (LDR)

Figure 12-14 shows the layout of the destination format register (DFR). The 4-bit model field in this register selects one of two models (flat or cluster) that can be used to interpret the MDA when using logical destination mode.

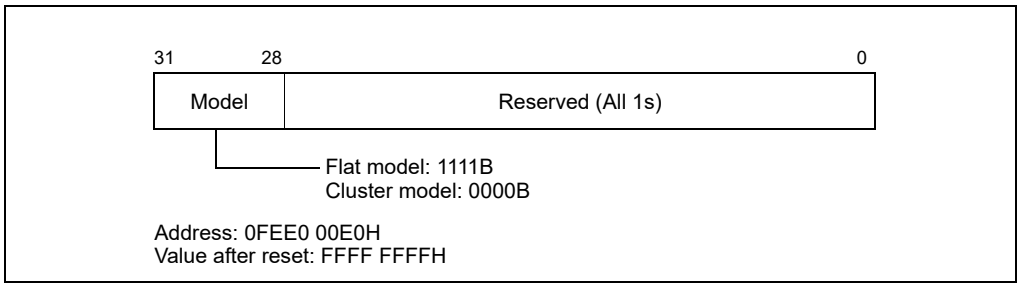


Figure 12-14. Destination Format Register (DFR)

The interpretation of MDA for the two models is described in the following paragraphs.

1. **Flat Model** — This model is selected by programming DFR bits 28 through 31 to 1111. Here, a unique logical APIC ID can be established for up to 8 local APICs by setting a different bit in the logical APIC ID field of the LDR for each local APIC. A group of local APICs can then be selected by setting one or more bits in the MDA.

   Each local APIC performs a bit-wise AND of the MDA and its logical APIC ID. If a true condition (non-zero) is detected, the local APIC accepts the IPI message. A broadcast to all APICs is achieved by setting the MDA to 1s.

2. **Cluster Model** — This model is selected by programming DFR bits 28 through 31 to 0000. This model supports two basic destination schemes: flat cluster and hierarchical cluster.

   The flat cluster destination model is only supported for P6 family and Pentium processors. Using this model, all APICs are assumed to be connected through the APIC bus. Bits 60 through 63 of the MDA contains the encoded address of the destination cluster and bits 56 through 59 identify up to four local APICs within the cluster (each bit is assigned to one local APIC in the cluster, as in the flat connection model). To identify one or more local APICs, bits 60 through 63 of the MDA are compared with bits 28 through 31 of the LDR to determine if a local APIC is part of the cluster. Bits 56 through 59 of the MDA are compared with Bits 24 through 27 of the LDR to identify a local APICs within the cluster.

   Sets of processors within a cluster can be specified by writing the target cluster address in bits 60 through 63 of the MDA and setting selected bits in bits 56 through 59 of the MDA, corresponding to the chosen members of the cluster. In this mode, 15 clusters (with cluster addresses of 0 through 14) each having 4 local APICs can be specified in the message. For the P6 and Pentium processor's local APICs, however, the APIC arbitration ID supports only 15 APIC agents. Therefore, the total number of processors and their local APICs supported in this mode is limited to 15. Broadcast to all local APICs is achieved by setting all destination bits to one. This guarantees a match on all clusters and selects all APICs in each cluster. A broadcast IPI or I/O subsystem broadcast interrupt with lowest priority delivery mode is not supported in cluster mode and must not be configured by software.

   The hierarchical cluster destination model can be used with Pentium 4, Intel Xeon, P6 family, or Pentium processors. With this model, a hierarchical network can be created by connecting different flat clusters via independent system or APIC buses. This scheme requires a cluster manager within each cluster, which is responsible for handling message passing between system or APIC buses. One cluster contains up to 4 agents. Thus 15 cluster managers, each with 4 agents, can form a network of up to 60 APIC agents. Note that hierar-

chical APIC networks requires a special cluster manager device, which is not part of the local or the I/O APIC units.

NOTES

All processors that have their APIC software enabled (using the spurious vector enable/disable bit) must have their DFRs (Destination Format Registers) programmed identically.

The default mode for DFR is flat mode. If you are using cluster mode, DFRs must be programmed before the APIC is software enabled. Since some chipsets do not accurately track a system view of the logical mode, program DFRs as soon as possible after starting the processor.

12.6.2.3 Broadcast/Self Delivery Mode

The destination shorthand field of the ICR allows the delivery mode to be by-passed in favor of broadcasting the IPI to all the processors on the system bus and/or back to itself (see Section 12.6.1, "Interrupt Command Register (ICR)"). Three destination shorthands are supported: self, all excluding self, and all including self. The destination mode is ignored when a destination shorthand is used.

12.6.2.4 Lowest Priority Delivery Mode

With lowest priority delivery mode, the ICR is programmed to send an IPI to several processors on the system bus, using the logical or shorthand destination mechanism for selecting the processor. The selected processors then arbitrate with one another over the system bus or the APIC bus, with the lowest-priority processor accepting the IPI.

For systems based on the Intel Xeon processor, the chipset bus controller accepts messages from the I/O APIC agents in the system and directs interrupts to the processors on the system bus. When using the lowest priority delivery mode, the chipset chooses a target processor to receive the interrupt out of the set of possible targets. The Pentium 4 processor provides a special bus cycle on the system bus that informs the chipset of the current task priority for each logical processor in the system. The chipset saves this information and uses it to choose the lowest priority processor when an interrupt is received.

For systems based on P6 family processors, the processor priority used in lowest-priority arbitration is contained in the arbitration priority register (APR) in each local APIC. Figure 12-15 shows the layout of the APR.

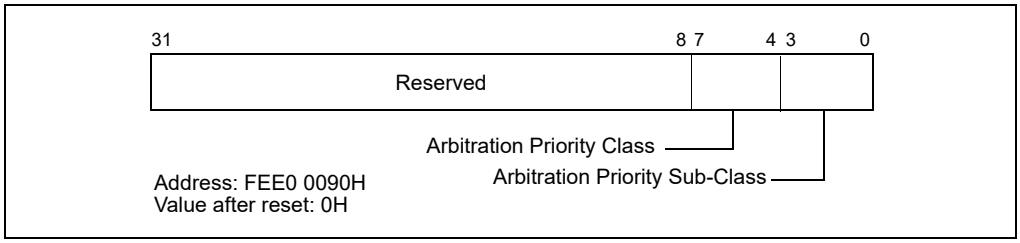


Figure 12-15. Arbitration Priority Register (APR)

The APR value is computed as follows:

```
IF (TPR[7:4] ≥ IRRV[7:4]) AND (TPR[7:4] > ISRV[7:4])
    THEN
        APR[7:0] ← TPR[7:0]
    ELSE
        APR[7:4] ← max(TPR[7:4] AND ISRV[7:4], IRRV[7:4])
        APR[3:0] ← 0.
```

Here, the TPR value is the task priority value in the TPR (see Figure 12-18), the IRRV value is the vector number for the highest priority bit that is set in the IRR (see Figure 12-20) or 00H (if no IRR bit is set), and the ISRV value is the vector number for the highest priority bit that is set in the ISR (see Figure 12-20). Following arbitration

among the destination processors, the processor with the lowest value in its APR handles the IPI and the other processors ignore it.

(P6 family and Pentium processors.) For these processors, if a **focus processor** exists**,** it may accept the interrupt, regardless of its priority. A processor is said to be the focus of an interrupt if it is currently servicing that interrupt or if it has a pending request for that interrupt. For Intel Xeon processors, the concept of a focus processor is not supported.

In operating systems that use the lowest priority delivery mode but do not update the TPR, the TPR information saved in the chipset will potentially cause the interrupt to be always delivered to the same processor from the logical set. This behavior is functionally backward compatible with the P6 family processor but may result in unexpected performance implications.

12.6.3 IPI Delivery and Acceptance

When the low double-word of the ICR is written to, the local APIC creates an IPI message from the information contained in the ICR and sends the message out on the system bus (Pentium 4 and Intel Xeon processors) or the APIC bus (P6 family and Pentium processors). The manner in which these IPIs are handled after being issued is described in Section 12.8, "Handling Interrupts."

12.7 SYSTEM AND APIC BUS ARBITRATION

When several local APICs and the I/O APIC are sending IPI and interrupt messages on the system bus (or APIC bus), the order in which the messages are sent and handled is determined through bus arbitration.

For the Pentium 4 and Intel Xeon processors, the local and I/O APICs use the arbitration mechanism defined for the system bus to determine the order in which IPIs are handled. This mechanism is non-architectural and cannot be controlled by software.

For the P6 family and Pentium processors, the local and I/O APICs use an APIC-based arbitration mechanism to determine the order in which IPIs are handled. Here, each local APIC is given an arbitration priority of from 0 to 15, which the I/O APIC uses during arbitration to determine which local APIC should be given access to the APIC bus. The local APIC with the highest arbitration priority always wins bus access. Upon completion of an arbitration round, the winning local APIC lowers its arbitration priority to 0 and the losing local APICs each raise theirs by 1.

The current arbitration priority for a local APIC is stored in a 4-bit, software-transparent arbitration ID (Arb ID) register. During reset, this register is initialized to the APIC ID number (stored in the local APIC ID register). The INIT level-deassert IPI, which is issued with and ICR command, can be used to resynchronize the arbitration priorities of the local APICs by resetting Arb ID register of each agent to its current APIC ID value. (The Pentium 4 and Intel Xeon processors do not implement the Arb ID register.)

Section 12.10, "APIC Bus Message Passing Mechanism and Protocol (P6 Family, Pentium Processors)," describes the APIC bus arbitration protocols and bus message formats, while Section 12.6.1, "Interrupt Command Register (ICR)," describes the INIT level de-assert IPI message.

Note that except for the SIPI IPI (see Section 12.6.1, "Interrupt Command Register (ICR)"), all bus messages that fail to be delivered to their specified destination or destinations are automatically retried. Software should avoid situations in which IPIs are sent to disabled or nonexistent local APICs, causing the messages to be resent repeatedly. Additionally, interrupt sources that target the APIC should be masked or changed to no longer target the APIC.

12.8 HANDLING INTERRUPTS

When a local APIC receives an interrupt from a local source, an interrupt message from an I/O APIC, or an IPI, the manner in which it handles the message depends on processor implementation, as described in the following sections.

12.8.1 Interrupt Handling with the Pentium 4 and Intel Xeon Processors

With the Pentium 4 and Intel Xeon processors, the local APIC handles the local interrupts, interrupt messages, and IPIs it receives as follows:

1. It determines if it is the specified destination or not (see Figure 12-16). If it is the specified destination, it accepts the message; if it is not, it discards the message.

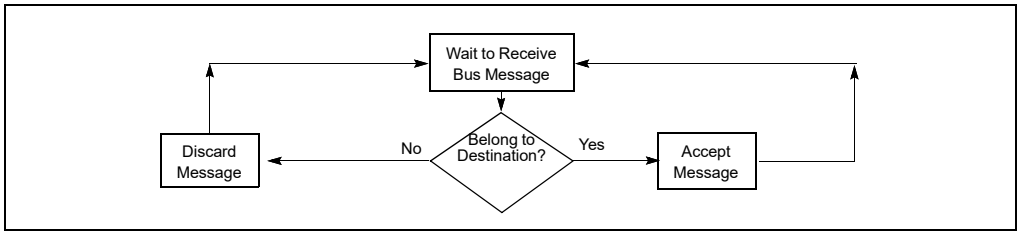


Figure 12-16. Interrupt Acceptance Flow Chart for the Local APIC (Pentium 4 and Intel Xeon Processors)

2. If the local APIC determines that it is the designated destination for the interrupt and if the interrupt request is an NMI, SMI, INIT, ExtINT, or SIPI, the interrupt is sent directly to the processor core for handling.
3. If the local APIC determines that it is the designated destination for the interrupt but the interrupt request is not one of the interrupts given in step 2, the local APIC sets the appropriate bit in the IRR.
4. When interrupts are pending in the IRR register, the local APIC dispatches them to the processor one at a time, based on their priority and the current processor priority in the PPR (see Section 12.8.3.1, "Task and Processor Priorities").
5. When a fixed interrupt has been dispatched to the processor core for handling, the completion of the handler routine is indicated with an instruction in the instruction handler code that writes to the end-of-interrupt (EOI) register in the local APIC (see Section 12.8.5, "Signaling Interrupt Servicing Completion"). The act of writing to the EOI register causes the local APIC to delete the interrupt from its ISR queue and (for level-triggered interrupts) send a message on the bus indicating that the interrupt handling has been completed. (A write to the EOI register must not be included in the handler routine for an NMI, SMI, INIT, ExtINT, or SIPI.)

12.8.2 Interrupt Handling with the P6 Family and Pentium Processors

With the P6 family and Pentium processors, the local APIC handles the local interrupts, interrupt messages, and IPIs it receives as follows (see Figure 12-17).

1. (IPIs only) The local APIC examines the IPI message to determines if it is the specified destination for the IPI as described in Section 12.6.2, "Determining IPI Destination." If it is the specified destination, it continues its acceptance procedure; if it is not the destination, it discards the IPI message. When the message specifies lowest-priority delivery mode, the local APIC will arbitrate with the other processors that were designated as recipients of the IPI message (see Section 12.6.2.4, "Lowest Priority Delivery Mode").
2. If the local APIC determines that it is the designated destination for the interrupt and if the interrupt request is an NMI, SMI, INIT, ExtINT, or INIT-deassert interrupt, or one of the MP protocol IPI messages (BIPI, FIPI, and SIPI), the interrupt is sent directly to the processor core for handling.
3. If the local APIC determines that it is the designated destination for the interrupt but the interrupt request is not one of the interrupts given in step 2, the local APIC looks for an open slot in one of its two pending interrupt queues contained in the IRR and ISR registers (see Figure 12-20). If a slot is available (see Section 12.8.4, "Interrupt Acceptance for Fixed Interrupts"), places the interrupt in the slot. If a slot is not available, it rejects the interrupt request and sends it back to the sender with a retry message.
4. When interrupts are pending in the IRR register, the local APIC dispatches them to the processor one at a time, based on their priority and the current processor priority in the PPR (see Section 12.8.3.1, "Task and Processor Priorities").
5. When a fixed interrupt has been dispatched to the processor core for handling, the completion of the handler routine is indicated with an instruction in the instruction handler code that writes to the end-of-interrupt (EOI)

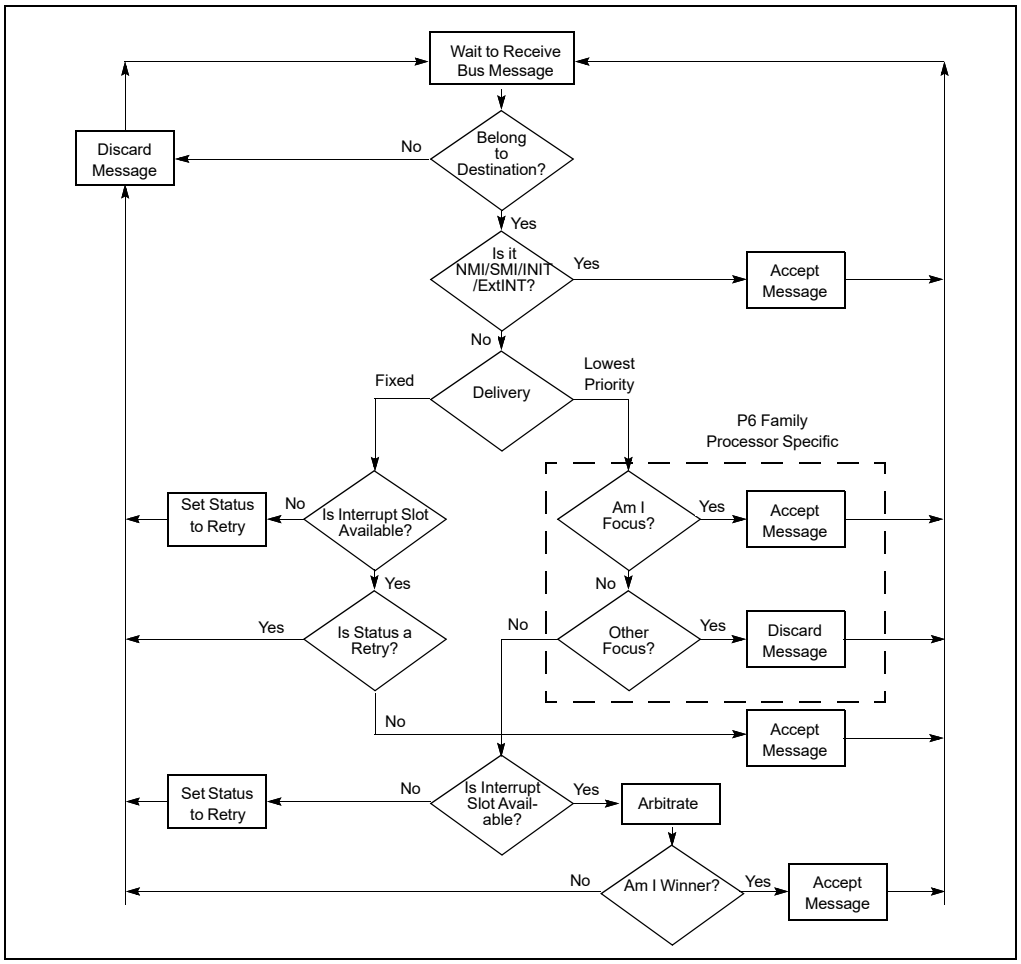


Figure 12-17. Interrupt Acceptance Flow Chart for the Local APIC (P6 Family and Pentium Processors)

register in the local APIC (see Section 12.8.5, "Signaling Interrupt Servicing Completion"). The act of writing to the EOI register causes the local APIC to delete the interrupt from its queue and (for level-triggered interrupts) send a message on the bus indicating that the interrupt handling has been completed. (A write to the EOI register must not be included in the handler routine for an NMI, SMI, INIT, ExtINT, or SIPI.)

The following sections describe the acceptance of interrupts and their handling by the local APIC and processor in greater detail.

12.8.3 Interrupt, Task, and Processor Priority

Each interrupt delivered to the processor through the local APIC has a priority based on its vector number. The local APIC uses this priority to determine when to service the interrupt relative to the other activities of the processor, including the servicing of other interrupts.

Each interrupt vector is an 8-bit value. The **interrupt-priority class** is the value of bits 7:4 of the interrupt vector. The lowest interrupt-priority class is 1 and the highest is 15; interrupts with vectors in the range 0–15 (with interrupt-priority class 0) are illegal and are never delivered. Because vectors 0–31 are reserved for dedicated uses by the Intel 64 and IA-32 architectures, software should configure interrupt vectors to use interrupt-priority classes in the range 2–15.

Each interrupt-priority class encompasses 16 vectors. The relative priority of interrupts within an interrupt-priority class is determined by the value of bits 3:0 of the vector number. The higher the value of those bits, the higher the

priority within that interrupt-priority class. Thus, each interrupt vector comprises two parts, with the high 4 bits indicating its interrupt-priority class and the low 4 bits indicating its ranking within the interrupt-priority class.

12.8.3.1 Task and Processor Priorities

The local APIC also defines a **task priority** and a **processor priority** that determine the order in which interrupts are handled. The **task-priority class** is the value of bits 7:4 of the task-priority register (TPR), which can be written by software (TPR is a read/write register); see Figure 12-18.

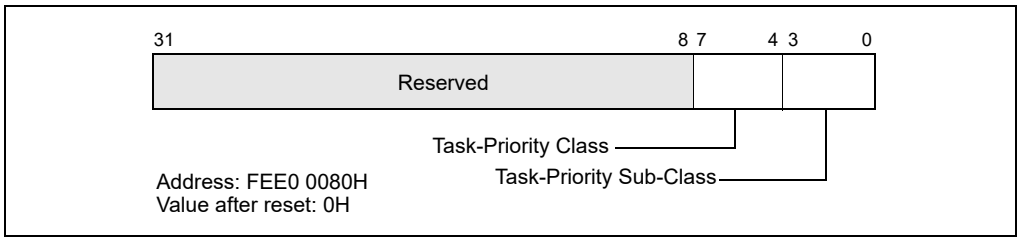


Figure 12-18. Task-Priority Register (TPR)

NOTE

In this discussion, the term "task" refers to a software defined task, process, thread, program, or routine that is dispatched to run on the processor by the operating system. It does not refer to an IA-32 architecture defined task as described in Chapter 9, "Task Management."

The task priority allows software to set a priority threshold for interrupting the processor. This mechanism enables the operating system to temporarily block low priority interrupts from disturbing high-priority work that the processor is doing. The ability to block such interrupts using task priority results from the way that the TPR controls the value of the processor-priority register (PPR).[1]

The **processor-priority class** is a value in the range 0–15 that is maintained in bits 7:4 of the processor-priority register (PPR); see Figure 12-19. The PPR is a read-only register. The processor-priority class represents the current priority at which the processor is executing.

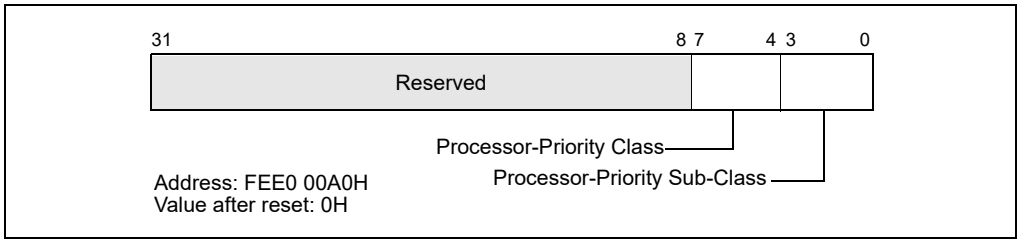


Figure 12-19. Processor-Priority Register (PPR)

The value of the PPR is based on the value of TPR and the value ISRV; ISRV is the vector number of the highest priority bit that is set in the ISR or 00H if no bit is set in the ISR. (See Section 12.8.4 for more details on the ISR.) The value of PPR is determined as follows:

- PPR[7:4] (the processor-priority class) the maximum of TPR[7:4] (the task- priority class) and ISRV[7:4] (the priority of the highest priority interrupt in service).
- PPR[3:0] (the processor-priority sub-class) is determined as follows:
  - — If TPR[7:4] > ISRV[7:4], PPR[3:0] is TPR[3:0] (the task-priority sub-class).
  - — If TPR[7:4] < ISRV[7:4], PPR[3:0] is 0.
  - — If TPR[7:4] = ISRV[7:4], PPR[3:0] may be either TPR[3:0] or 0. The actual behavior is model-specific.

1. The TPR also determines the arbitration priority of the local processor; see Section 12.6.2.4, "Lowest Priority Delivery Mode."

The processor-priority class determines the priority threshold for interrupting the processor. The processor will deliver only those interrupts that have an interrupt-priority class higher than the processor-priority class in the PPR. If the processor-priority class is 0, the PPR does not inhibit the delivery any interrupt; if it is 15, the processor inhibits the delivery of all interrupts. (The processor-priority mechanism does not affect the delivery of interrupts with the NMI, SMI, INIT, ExtINT, INIT-deassert, and start-up delivery modes.)

The processor does not use the processor-priority sub-class to determine which interrupts to deliver and which to inhibit. (The processor uses the processor-priority sub-class only to satisfy reads of the PPR.)

12.8.4 Interrupt Acceptance for Fixed Interrupts

The local APIC queues the fixed interrupts that it accepts in one of two interrupt pending registers: the interrupt request register (IRR) or in-service register (ISR). These two 256-bit read-only registers are shown in Figure 12-20. The 256 bits in these registers represent the 256 possible vectors; vectors 0 through 15 are reserved by the APIC (see also: Section 12.5.2, “Valid Interrupt Vectors”).

NOTE

All interrupts with an NMI, SMI, INIT, ExtINT, start-up, or INIT-deassert delivery mode bypass the IRR and ISR registers and are sent directly to the processor core for servicing.

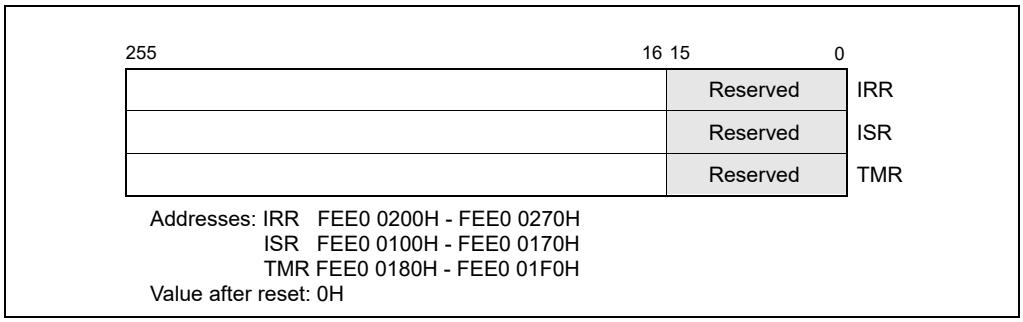


Figure 12-20. IRR, ISR, and TMR Registers

The IRR contains the active interrupt requests that have been accepted, but not yet dispatched to the processor for servicing. When the local APIC accepts an interrupt, it sets the bit in the IRR that corresponds the vector of the accepted interrupt. When the processor core is ready to handle the next interrupt, the local APIC clears the highest priority IRR bit that is set and sets the corresponding ISR bit. The vector for the highest priority bit set in the ISR is then dispatched to the processor core for servicing.

While the processor is servicing the highest priority interrupt, the local APIC can send additional fixed interrupts by setting bits in the IRR. When the interrupt service routine issues a write to the EOI register (see Section 12.8.5, “Signaling Interrupt Servicing Completion”), the local APIC responds by clearing the highest priority ISR bit that is set. It then repeats the process of clearing the highest priority bit in the IRR and setting the corresponding bit in the ISR. The processor core then begins executing the service routing for the highest priority bit set in the ISR.

If more than one interrupt is generated with the same vector number, the local APIC can set the bit for the vector both in the IRR and the ISR. This means that for the Pentium 4 and Intel Xeon processors, the IRR and ISR can queue two interrupts for each interrupt vector: one in the IRR and one in the ISR. Any additional interrupts issued for the same interrupt vector are collapsed into the single bit in the IRR.

For the P6 family and Pentium processors, the IRR and ISR registers can queue no more than two interrupts per interrupt vector and will reject other interrupts that are received within the same vector.

If the local APIC receives an interrupt with an interrupt-priority class higher than that of the interrupt currently in service, and interrupts are enabled in the processor core, the local APIC dispatches the higher priority interrupt to the processor immediately (without waiting for a write to the EOI register). The currently executing interrupt handler is then interrupted so the higher-priority interrupt can be handled. When the handling of the higher-priority interrupt has been completed, the servicing of the interrupted interrupt is resumed.

The trigger mode register (TMR) indicates the trigger mode of the interrupt (see Figure 12-20). Upon acceptance of an interrupt into the IRR, the corresponding TMR bit is cleared for edge-triggered interrupts and set for level-triggered interrupts. If a TMR bit is set when an EOI cycle for its corresponding interrupt vector is generated, an EOI message is sent to all I/O APICs.

12.8.5 Signaling Interrupt Servicing Completion

For all interrupts except those delivered with the NMI, SMI, INIT, ExtINT, the start-up, or INIT-Deassert delivery mode, the interrupt handler must include a write to the end-of-interrupt (EOI) register (see Figure 12-21). This write must occur at the end of the handler routine, sometime before the IRET instruction. This action indicates that the servicing of the current interrupt is complete and the local APIC can issue the next interrupt from the ISR.

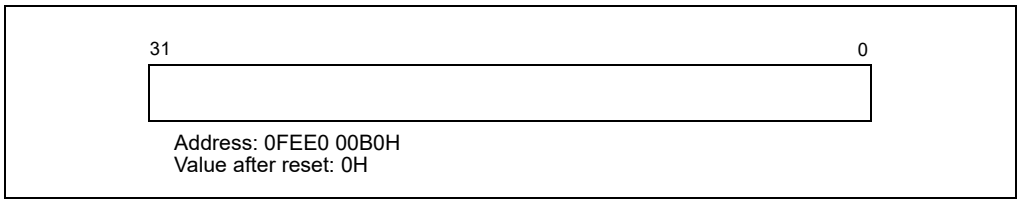


Figure 12-21. EOI Register

Upon receiving an EOI, the APIC clears the highest priority bit in the ISR and dispatches the next highest priority interrupt to the processor. If the terminated interrupt was a level-triggered interrupt, the local APIC also sends an end-of-interrupt message to all I/O APICs.

System software may prefer to direct EOIs to specific I/O APICs rather than having the local APIC send end-of-interrupt messages to all I/O APICs.

Software can inhibit the broadcast of EOI message by setting bit 12 of the Spurious Interrupt Vector Register (see Section 12.9). If this bit is set, a broadcast EOI is not generated on an EOI cycle even if the associated TMR bit indicates that the current interrupt was level-triggered. The default value for the bit is 0, indicating that EOI broadcasts are performed.

Bit 12 of the Spurious Interrupt Vector Register is reserved to 0 if the processor does not support suppression of EOI broadcasts. Support for EOI-broadcast suppression is reported in bit 24 in the Local APIC Version Register (see Section 12.4.8); the feature is supported if that bit is set to 1. When supported, the feature is available in both xAPIC mode and x2APIC mode.

System software desiring to perform directed EOIs for level-triggered interrupts should set bit 12 of the Spurious Interrupt Vector Register and follow each the EOI to the local xAPIC for a level triggered interrupt with a directed EOI to the I/O APIC generating the interrupt (this is done by writing to the I/O APIC's EOI register). System software performing directed EOIs must retain a mapping associating level-triggered interrupts with the I/O APICs in the system.

12.8.6 Task Priority in IA-32e Mode

In IA-32e mode, operating systems can manage the 16 interrupt-priority classes (see Section 12.8.3, "Interrupt, Task, and Processor Priority") explicitly using the task priority register (TPR). Operating systems can use the TPR to temporarily block specific (low-priority) interrupts from interrupting a high-priority task. This is done by loading TPR with a value in which the task-priority class corresponds to the highest interrupt-priority class that is to be blocked. For example:

- Loading the TPR with a task-priority class of 8 (01000B) blocks all interrupts with an interrupt-priority class of 8 or less while allowing all interrupts with an interrupt-priority class of 9 or more to be recognized.
- Loading the TPR with a task-priority class of 0 enables all external interrupts.
- Loading the TPR with a task-priority class of 0FH (01111B) disables all external interrupts.

The TPR (shown in Figure 12-18) is cleared to 0 on reset. In 64-bit mode, software can read and write the TPR using an alternate interface, MOV CR8 instruction. The new task-priority class is established when the MOV CR8

instruction completes execution. Software does not need to force serialization after loading the TPR using MOV CR8.

Use of the MOV CRn instruction requires a privilege level of 0. Programs running at privilege level greater than 0 cannot read or write the TPR. An attempt to do so causes a general-protection exception. The TPR is abstracted from the interrupt controller (IC), which prioritizes and manages external interrupt delivery to the processor. The IC can be an external device, such as an APIC or 8259. Typically, the IC provides a priority mechanism similar or identical to the TPR. The IC, however, is considered implementation-dependent with the under-lying priority mechanisms subject to change. CR8, by contrast, is part of the Intel 64 architecture. Software can depend on this definition remaining unchanged.

Figure 12-22 shows the layout of CR8; only the low four bits are used. The remaining 60 bits are reserved and must be written with zeros. Failure to do this causes a general-protection exception.

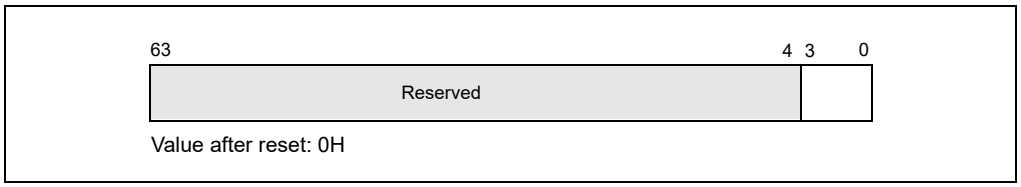


Figure 12-22. CR8 Register

12.8.6.1 Interaction of Task Priorities between CR8 and APIC

The first implementation of Intel 64 architecture includes a local advanced programmable interrupt controller (APIC) that is similar to the APIC used with previous IA-32 processors. Some aspects of the local APIC affect the operation of the architecturally defined task priority register and the programming interface using CR8.

Notable CR8 and APIC interactions are:

- The processor powers up with the local APIC enabled.
- The APIC must be enabled for CR8 to function as the TPR. Writes to CR8 are reflected into the APIC Task Priority Register.
- APIC.TPR[bits 7:4] = CR8[bits 3:0], APIC.TPR[bits 3:0] = 0. A read of CR8 returns a 64-bit value which is the value of TPR[bits 7:4], zero extended to 64 bits.

There are no ordering mechanisms between direct updates of the APIC.TPR and CR8. Operating software should implement either direct APIC TPR updates or CR8 style TPR updates but not mix them. Software can use a serializing instruction (for example, CPUID) to serialize updates between MOV CR8 and stores to the APIC.

12.9 SPURIOUS INTERRUPT

A special situation may occur when a processor raises its task priority to be greater than or equal to the level of the interrupt for which the processor INTR signal is currently being asserted. If at the time the INTA cycle is issued, the interrupt that was to be dispensed has become masked (programmed by software), the local APIC will deliver a spurious-interrupt vector. Dispensing the spurious-interrupt vector does not affect the ISR, so the handler for this vector should return without an EOI.

The vector number for the spurious-interrupt vector is specified in the spurious-interrupt vector register (see Figure 12-23). The functions of the fields in this register are as follows:

Spurious Vector Determines the vector number to be delivered to the processor when the local APIC generates a spurious vector.

(Pentium 4 and Intel Xeon processors.) Bits 0 through 7 of the this field are programmable by software.

(P6 family and Pentium processors). Bits 4 through 7 of the this field are programmable by software, and bits 0 through 3 are hardwired to logical ones. Software writes to bits 0 through 3 have no effect.

APIC Software Enable/Disable

Allows software to temporarily enable (1) or disable (0) the local APIC (see Section 12.4.3, "Enabling or Disabling the Local APIC").

Focus Processor Checking

Determines if focus processor checking is enabled (0) or disabled (1) when using the lowest-priority delivery mode. In Pentium 4 and Intel Xeon processors, this bit is reserved and should be cleared to 0.

Suppress EOI Broadcasts

Determines whether an EOI for a level-triggered interrupt causes EOI messages to be broadcast to the I/O APICs (0) or not (1). See Section 12.8.5. The default value for this bit is 0, indicating that EOI broadcasts are performed. This bit is reserved to 0 if the processor does not support EOI-broadcast suppression.

NOTE

Do not program an LVT or IOAPIC RTE with a spurious vector even if you set the mask bit. A spurious vector ISR does not do an EOI. If for some reason an interrupt is generated by an LVT or RTE entry, the bit in the in-service register will be left set for the spurious vector. This will mask all interrupts at the same or lower priority

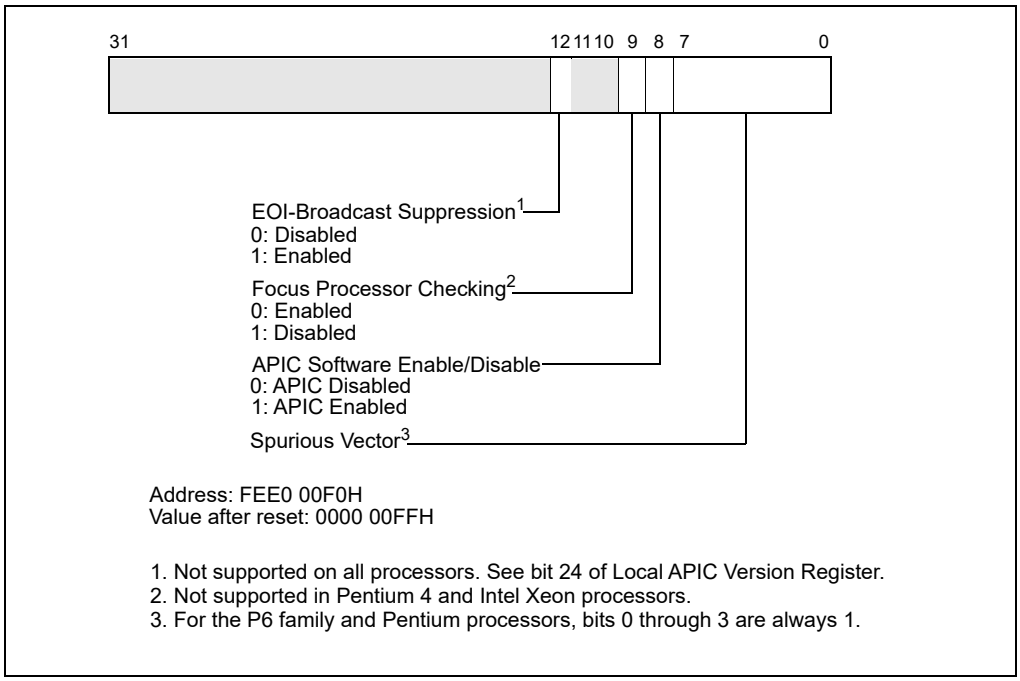


Figure 12-23. Spurious-Interrupt Vector Register (SVR)

12.10 APIC BUS MESSAGE PASSING MECHANISM AND PROTOCOL (P6 FAMILY, PENTIUM PROCESSORS)

The Pentium 4 and Intel Xeon processors pass messages among the local and I/O APICs on the system bus, using the system bus message passing mechanism and protocol.

The P6 family and Pentium processors, pass messages among the local and I/O APICs on the serial APIC bus, as follows. Because only one message can be sent at a time on the APIC bus, the I/O APIC and local APICs employ a "rotating priority" arbitration protocol to gain permission to send a message on the APIC bus. One or more APICs may start sending their messages simultaneously. At the beginning of every message, each APIC presents the type of the message it is sending and its current arbitration priority on the APIC bus. This information is used for arbitration. After each arbitration cycle (within an arbitration round), only the potential winners keep driving the bus.

By the time all arbitration cycles are completed, there will be only one APIC left driving the bus. Once a winner is selected, it is granted exclusive use of the bus, and will continue driving the bus to send its actual message.

After each successfully transmitted message, all APICs increase their arbitration priority by 1. The previous winner (that is, the one that has just successfully transmitted its message) assumes a priority of 0 (lowest). An agent whose arbitration priority was 15 (highest) during arbitration, but did not send a message, adopts the previous winner's arbitration priority, incremented by 1.

Note that the arbitration protocol described above is slightly different if one of the APICs issues a special End-Of-Interrupt (EOI). This high-priority message is granted the bus regardless of its sender's arbitration priority, unless more than one APIC issues an EOI message simultaneously. In the latter case, the APICs sending the EOI messages arbitrate using their arbitration priorities.

If the APICs are set up to use "lowest priority" arbitration (see Section 12.6.2.4, "Lowest Priority Delivery Mode") and multiple APICs are currently executing at the lowest priority (the value in the APR register), the arbitration priorities (unique values in the Arb ID register) are used to break ties. All 8 bits of the APR are used for the lowest priority arbitration.

12.10.1 Bus Message Formats

See Section 12.13, "APIC Bus Message Formats," for a description of bus message formats used to transmit messages on the serial APIC bus.

12.11 MESSAGE SIGNALLED INTERRUPTS

The PCI Local Bus Specification, Rev 2.2 (www.pcisig.com) introduces the concept of message signalled interrupts. As the specification indicates:

> "Message signalled interrupts (MSI) is an optional feature that enables PCI devices to request service by writing a system-specified message to a system-specified address (PCI DWORD memory write transaction). The transaction address specifies the message destination while the transaction data specifies the message. System software is expected to initialize the message destination and message during device configuration, allocating one or more non-shared messages to each MSI capable function."

The capabilities mechanism provided by the PCI Local Bus Specification is used to identify and configure MSI capable PCI devices. Among other fields, this structure contains a Message Data Register and a Message Address Register. To request service, the PCI device function writes the contents of the Message Data Register to the address contained in the Message Address Register (and the Message Upper Address register for 64-bit message addresses).

Section 12.11.1 and Section 12.11.2 provide layout details for the Message Address Register and the Message Data Register. The operation issued by the device is a PCI write command to the Message Address Register with the Message Data Register contents. The operation follows semantic rules as defined for PCI write operations and is a DWORD operation.

12.11.1 Message Address Register Format

The format of the Message Address Register (lower 32-bits) is shown in Figure 12-24.

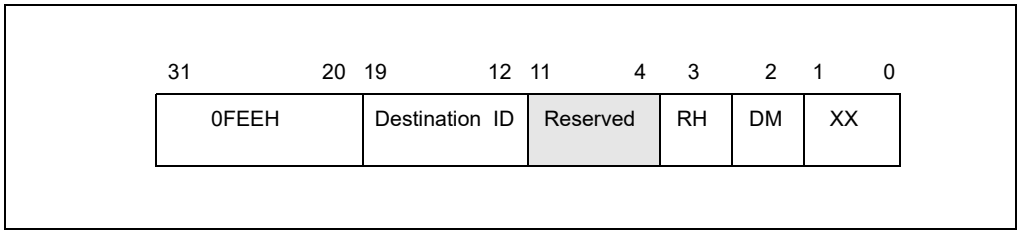


Figure 12-24. Layout of the MSI Message Address Register

Fields in the Message Address Register are as follows:

1. **Bits 31-20** — These bits contain a fixed value for interrupt messages (0FEEH). This value locates interrupts at the 1-MByte area with a base address of 4G – 18M. All accesses to this region are directed as interrupt messages. Care must to be taken to ensure that no other device claims the region as I/O space.
2. **Destination ID** — This field contains an 8-bit destination ID. It identifies the message's target processor(s). The destination ID corresponds to bits 63:56 of the I/O APIC Redirection Table Entry if the IOAPIC is used to dispatch the interrupt to the processor(s).
3. **Redirection hint indication** (RH) — When this bit is set, the message is directed to the processor with the lowest interrupt priority among processors that can receive the interrupt.
    - When RH is 0, the interrupt is directed to the processor listed in the Destination ID field.
    - When RH is 1 and the physical destination mode is used, the Destination ID field must not be set to FFH; it must point to a processor that is present and enabled to receive the interrupt.
    - When RH is 1 and the logical destination mode is active in a system using a flat addressing model, the Destination ID field must be set so that bits set to 1 identify processors that are present and enabled to receive the interrupt.
    - If RH is set to 1 and the logical destination mode is active in a system using cluster addressing model, then Destination ID field must not be set to FFH; the processors identified with this field must be present and enabled to receive the interrupt.
4. **Destination mode (DM)** — This bit indicates whether the Destination ID field should be interpreted as logical or physical APIC ID for delivery of the lowest priority interrupt.
    - If RH is 1 and DM is 0, the Destination ID field is in physical destination mode and only the processor in the system that has the matching APIC ID is considered for delivery of that interrupt (this means no re-direction).
    - If RH is 1 and DM is 1, the Destination ID Field is interpreted as in logical destination mode and the redirection is limited to only those processors that are part of the logical group of processors based on the processor's logical APIC ID and the Destination ID field in the message. The logical group of processors consists of those identified by matching the 8-bit Destination ID with the logical destination identified by the Destination Format Register and the Logical Destination Register in each local APIC. The details are similar to those described in Section 12.6.2, "Determining IPI Destination."
    - If RH is 0, then the DM bit is ignored and the message is sent ahead independent of whether the physical or logical destination mode is used.

12.11.2 Message Data Register Format

The layout of the Message Data Register is shown in Figure 12-25.

Reserved fields are not assumed to be any value. Software must preserve their contents on writes. Other fields in the Message Data Register are described below.

1. **Vector** — This 8-bit field contains the interrupt vector associated with the message. Values range from 010H to 0FEH. Software must guarantee that the field is not programmed with vector 00H to 0FH.
2. **Delivery Mode** — This 3-bit field specifies how the interrupt receipt is handled. Delivery Modes operate only in conjunction with specified Trigger Modes. Correct Trigger Modes must be guaranteed by software. Restrictions are indicated below:
    a. **000B (Fixed Mode)** — Deliver the signal to all the agents listed in the destination. The Trigger Mode for fixed delivery mode can be edge or level.
    b. **001B (Lowest Priority)** — Deliver the signal to the agent that is executing at the lowest priority of all agents listed in the destination field. The trigger mode can be edge or level.
    c. **010B (System Management Interrupt or SMI)** — The delivery mode is edge only. For systems that rely on SMI semantics, the vector field is ignored but must be programmed to all zeroes for future compatibility.

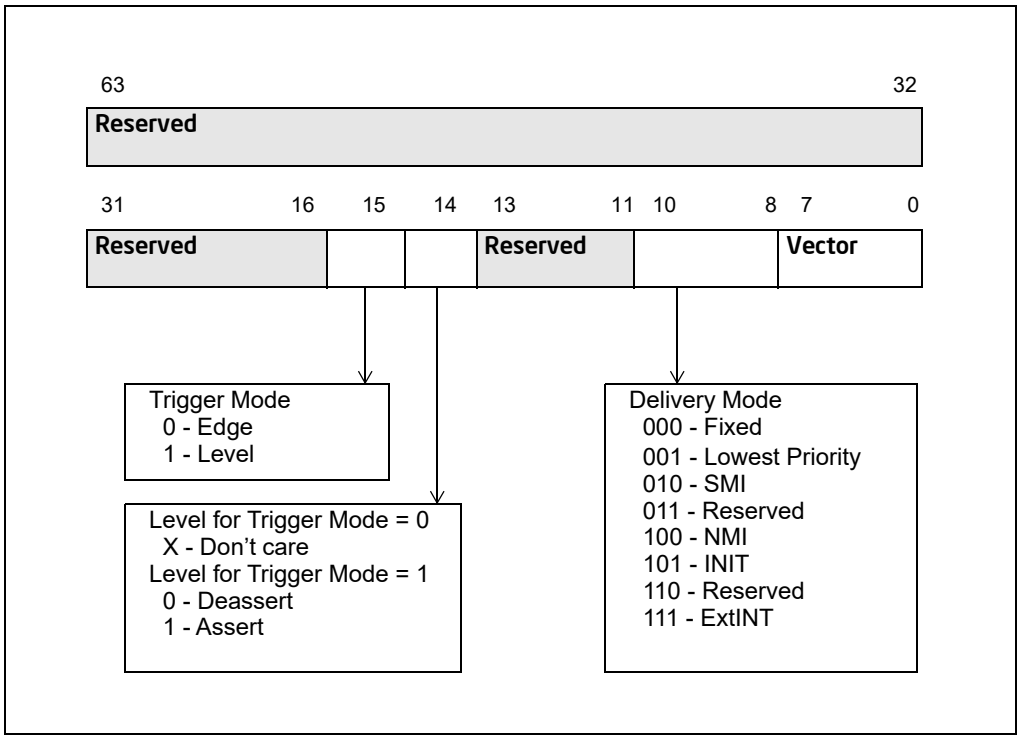


Figure 12-25. Layout of the MSI Message Data Register

d. **100B (NMI)** — Deliver the signal to all the agents listed in the destination field. The vector information is ignored. NMI is an edge triggered interrupt regardless of the Trigger Mode Setting.

e. **101B (INIT)** — Deliver this signal to all the agents listed in the destination field. The vector information is ignored. INIT is an edge triggered interrupt regardless of the Trigger Mode Setting.

f. **111B (ExtINT)** — Deliver the signal to the INTR signal of all agents in the destination field (as an interrupt that originated from an 8259A compatible interrupt controller). The vector is supplied by the INTA cycle issued by the activation of the ExtINT. ExtINT is an edge triggered interrupt.

3. **Level** — Edge triggered interrupt messages are always interpreted as assert messages. For edge triggered interrupts this field is not used. For level triggered interrupts, this bit reflects the state of the interrupt input.

4. **Trigger Mode** — This field indicates the signal type that will trigger a message.

   a. **0** — Indicates edge sensitive.

   b. **1** — Indicates level sensitive.

12.12 EXTENDED XAPIC (X2APIC)

The x2APIC architecture extends the xAPIC architecture (described in Section 12.4) in a backward compatible manner and provides forward extendability for future Intel platform innovations. Specifically, the x2APIC architecture does the following.

- Retains all key elements of compatibility to the xAPIC architecture.
  - — Delivery modes.
  - — Interrupt and processor priorities.
  - — Interrupt sources.
  - — Interrupt destination types.
- Provides extensions to scale processor addressability for both the logical and physical destination modes.

- Adds new features to enhance performance of interrupt delivery.
- Reduces complexity of logical destination mode interrupt delivery on link based platform architectures.
- Uses MSR programming interface to access APIC registers in x2APIC mode instead of memory-mapped interfaces. Memory-mapped interface is supported when operating in xAPIC mode.

12.12.1 Detecting and Enabling x2APIC Mode

Processor support for x2APIC mode can be detected by executing CPUID with EAX=1 and then checking ECX, bit 21 ECX. If CPUID.(EAX=1):ECX.21 is set , the processor supports the x2APIC capability and can be placed into the x2APIC mode.

System software can place the local APIC in the x2APIC mode by setting the x2APIC mode enable bit (bit 10) in the IA32_APIC_BASE MSR at MSR address 01BH. The layout for the IA32_APIC_BASE MSR is shown in Figure 12-26.

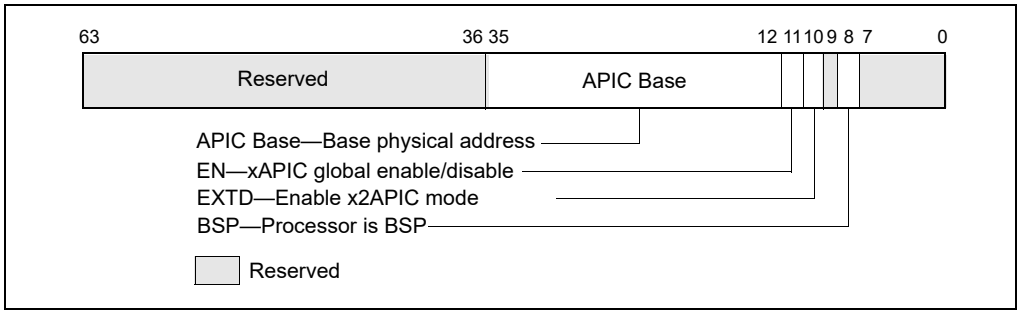


Figure 12-26. IA32_APIC_BASE MSR Supporting x2APIC

Table 12-5, "x2APIC operating mode configurations" describe the possible combinations of the enable bit (EN - bit 11) and the extended mode bit (EXTD - bit 10) in the IA32_APIC_BASE MSR.

Table 12-5. x2APIC Operating Mode Configurations

xAPIC global enable (IA32_APIC_BASE[11])	x2APIC enable (IA32_APIC_BASE[10])	Description
0	0	local APIC is disabled
0	1	Invalid
1	0	local APIC is enabled in xAPIC mode
1	1	local APIC is enabled in x2APIC mode

Once the local APIC has been switched to x2APIC mode (EN = 1, EXTD = 1), switching back to xAPIC mode would require system software to disable the local APIC unit. Specifically, attempting to write a value to the IA32_APIC_BASE MSR that has (EN= 1, EXTD = 0) when the local APIC is enabled and in x2APIC mode causes a general-protection exception. Once bit 10 in IA32_APIC_BASE MSR is set, the only way to leave x2APIC mode using IA32_APIC_BASE would require a WRMSR to set both bit 11 and bit 10 to zero. Section 12.12.5, "x2APIC State Transitions," provides a detailed state diagram for the state transitions allowed for the local APIC.

12.12.1.1 Instructions to Access APIC Registers

In x2APIC mode, system software uses RDMSR and WRMSR to access the APIC registers. The MSR addresses for accessing the x2APIC registers are architecturally defined and specified in Section 12.12.1.2, "x2APIC Register Address Space." Executing the RDMSR instruction with the APIC register address specified in ECX returns the content of bits 0 through 31 of the APIC registers in EAX. Bits 32 through 63 are returned in register EDX - these bits are reserved if the APIC register being read is a 32-bit register. Similarly executing the WRMSR instruction with the APIC register address in ECX, writes bits 0 to 31 of register EAX to bits 0 to 31 of the specified APIC register. If the register is a 64-bit register then bits 0 to 31 of register EDX are written to bits 32 to 63 of the APIC register. The

Interrupt Command Register is the only APIC register that is implemented as a 64-bit MSR. The semantics of handling reserved bits are defined in Section 12.12.1.3, "Reserved Bit Checking."

12.12.1.2 x2APIC Register Address Space

The MSR address range 800H through 8FFH is architecturally reserved and dedicated for accessing APIC registers in x2APIC mode. Table 12-6 lists the APIC registers that are available in x2APIC mode. When appropriate, the table also gives the offset at which each register is available on the page referenced by IA32_APIC_BASE[35:12] in xAPIC mode.

There is a one-to-one mapping between the x2APIC MSRs and the legacy xAPIC register offsets with the following exceptions:

- The Destination Format Register (DFR): The DFR, supported at offset 0E0H in xAPIC mode, is not supported in x2APIC mode. There is no MSR with address 80EH.
- The Interrupt Command Register (ICR): The two 32-bit registers in xAPIC mode (at offsets 300H and 310H) are merged into a single 64-bit MSR in x2APIC mode (with MSR address 830H). There is no MSR with address 831H.
- The SELF IPI register. This register is available only in x2APIC mode at address 83FH. In xAPIC mode, there is no register defined at offset 3F0H.

MSR addresses in the range 800H–8FFH that are not listed in Table 12-6 (including 80EH and 831H) are reserved. Executions of RDMSR and WRMSR that attempt to access such addresses cause general-protection exceptions.

The MSR address space is compressed to allow for future growth. Every 32 bit register on a 128-bit boundary in the legacy MMIO space is mapped to a single MSR in the local x2APIC MSR address space. The upper 32-bits of all x2APIC MSRs (except for the ICR) are reserved.

Table 12-6. Local APIC Register Address Map Supported by x2APIC

MSR Address (x2APIC mode)	MMIO Offset (xAPIC mode)	Register Name	MSR R/W Semantics	Comments
802H	020H	Local APIC ID register	Read-only[1]	See Section 12.12.5.1 for initial values.
803H	030H	Local APIC Version register	Read-only	Same version used in xAPIC mode and x2APIC mode.
808H	080H	Task Priority Register (TPR)	Read/write	Bits 31:8 are reserved.[2]
80AH	0A0H	Processor Priority Register (PPR)	Read-only	
80BH	0B0H	EOI register	Write-only[3]	WRMSR of a non-zero value causes #GP(0).
80DH	0D0H	Logical Destination Register (LDR)	Read-only	Read/write in xAPIC mode.
80FH	0F0H	Spurious Interrupt Vector Register (SVR)	Read/write	See Section 12.9 for reserved bits.
810H	100H	In-Service Register (ISR); bits 31:0	Read-only	
811H	110H	ISR bits 63:32	Read-only	
812H	120H	ISR bits 95:64	Read-only	
813H	130H	ISR bits 127:96	Read-only	
814H	140H	ISR bits 159:128	Read-only	
815H	150H	ISR bits 191:160	Read-only	
816H	160H	ISR bits 223:192	Read-only	

Table 12-6. Local APIC Register Address Map Supported by x2APIC (Contd.)

MSR Address (x2APIC mode)	MMIO Offset (xAPIC mode)	Register Name	MSR R/W Semantics	Comments
817H	170H	ISR bits 255:224	Read-only	
818H	180H	Trigger Mode Register (TMR); bits 31:0	Read-only	
819H	190H	TMR bits 63:32	Read-only	
81AH	1A0H	TMR bits 95:64	Read-only	
81BH	1B0H	TMR bits 127:96	Read-only	
81CH	1C0H	TMR bits 159:128	Read-only	
81DH	1D0H	TMR bits 191:160	Read-only	
81EH	1E0H	TMR bits 223:192	Read-only	
81FH	1F0H	TMR bits 255:224	Read-only	
820H	200H	Interrupt Request Register (IRR); bits 31:0	Read-only	
821H	210H	IRR bits 63:32	Read-only	
822H	220H	IRR bits 95:64	Read-only	
823H	230H	IRR bits 127:96	Read-only	
824H	240H	IRR bits 159:128	Read-only	
825H	250H	IRR bits 191:160	Read-only	
826H	260H	IRR bits 223:192	Read-only	
827H	270H	IRR bits 255:224	Read-only	
828H	280H	Error Status Register (ESR)	Read/write	WRMSR of a non-zero value causes #GP(0). See Section 12.5.3.
82FH	2F0H	LVT CMCI register	Read/write	See Figure 12-8 for reserved bits.
830H[4]	300H and 310H	Interrupt Command Register (ICR)	Read/write	See Figure 12-28 for reserved bits
832H	320H	LVT Timer register	Read/write	See Figure 12-8 for reserved bits.
833H	330H	LVT Thermal Sensor register	Read/write	See Figure 12-8 for reserved bits.
834H	340H	LVT Performance Monitoring register	Read/write	See Figure 12-8 for reserved bits.
835H	350H	LVT LINT0 register	Read/write	See Figure 12-8 for reserved bits.
836H	360H	LVT LINT1 register	Read/write	See Figure 12-8 for reserved bits.
837H	370H	LVT Error register	Read/write	See Figure 12-8 for reserved bits.
838H	380H	Initial Count register (for Timer)	Read/write	
839H	390H	Current Count register (for Timer)	Read-only	
83EH	3E0H	Divide Configuration Register (DCR; for Timer)	Read/write	See Figure 12-10 for reserved bits.
83FH	Not available	SELF IPI[5]	Write-only	Available only in x2APIC mode.

NOTES:

1. WRMSR causes #GP(0) for read-only registers.

2. WRMSR causes #GP(0) for attempts to set a reserved bit to 1 in a read/write register (including bits 63:32 of each register).
3. RDMSR causes #GP(0) for write-only registers.
4. MSR 831H is reserved; read/write operations cause general-protection exceptions. The contents of the APIC register at MMIO offset 310H are accessible in x2APIC mode through the MSR at address 830H.
5. SELF IPI register is supported only in x2APIC mode.

12.12.1.3 Reserved Bit Checking

Section 12.12.1.2 and Table 12-6 specifies the reserved bit definitions for the APIC registers in x2APIC mode. Non-zero writes (by WRMSR instruction) to reserved bits to these registers will raise a general protection fault exception while reads return zeros (RsvdZ semantics).

In x2APIC mode, the local APIC ID register is increased to 32 bits wide. This enables $2^{32}-1$ processors to be addressable in physical destination mode. This 32-bit value is referred to as "x2APIC ID". A processor implementation may choose to support less than 32 bits in its hardware. System software should be agnostic to the actual number of bits that are implemented. All non-implemented bits will return zeros on reads by software.

The APIC ID value of FFFF_FFFFH and the highest value corresponding to the implemented bit-width of the local APIC ID register in the system are reserved and cannot be assigned to any logical processor.

In x2APIC mode, the local APIC ID register is a read-only register to system software and will be initialized by hardware. It is accessed via the RDMSR instruction reading the MSR at address 0802H.

Each logical processor in the system (including clusters with a communication fabric) must be configured with an unique x2APIC ID to avoid collisions of x2APIC IDs. On DP and high-end MP processors targeted to specific market segments and depending on the system configuration, it is possible that logical processors in different and "unconnected" clusters power up initialized with overlapping x2APIC IDs. In these configurations, a model-specific means may be provided in those product segments to enable BIOS and/or platform firmware to re-configure the x2APIC IDs in some clusters to provide for unique and non-overlapping system wide IDs before configuring the disconnected components into a single system.

12.12.2 x2APIC Register Availability

The local APIC registers can be accessed via the MSR interface only when the local APIC has been switched to the x2APIC mode as described in Section 12.12.1. Accessing any APIC register in the MSR address range 0800H through 08FFH via RDMSR or WRMSR when the local APIC is not in x2APIC mode causes a general-protection exception. In x2APIC mode, the memory mapped interface is not available and any access to the MMIO interface will behave similar to that of a legacy xAPIC in globally disabled state. Table 12-7 provides the interactions between the legacy & extended modes and the legacy and register interfaces.

Table 12-7. MSR/MMIO Interface of a Local x2APIC in Different Modes of Operation

	MMIO Interface	MSR Interface
xAPIC mode	Available	General-protection exception
x2APIC mode	Behavior identical to xAPIC in globally disabled state	Available

12.12.3 MSR Access in x2APIC Mode

To allow for efficient access to the APIC registers in x2APIC mode, the serializing semantics of WRMSR are relaxed when writing to the APIC registers. Thus, system software should not use "WRMSR to APIC registers in x2APIC mode" as a serializing instruction. Read and write accesses to the APIC registers will occur in program order. A WRMSR to an APIC register may complete before all preceding stores are globally visible; software can prevent this by inserting a serializing instruction or the sequence MFENCE;LFENCE before the WRMSR.

The RDMSR instruction is not serializing and this behavior is unchanged when reading APIC registers in x2APIC mode. System software accessing the APIC registers using the RDMSR instruction should not expect a serializing behavior. (Note: The MMIO-based xAPIC interface is mapped by system software as an un-cached region. Consequently, read/writes to the xAPIC-MMIO interface have serializing semantics in the xAPIC mode.)

12.12.4 VM-Exit Controls for MSRs and x2APIC Registers

The VMX architecture allows a VMM to specify lists of MSRs to be loaded or stored on VMX transitions using the VMX-transition MSR areas (see VM-exit MSR-store address field, VM-exit MSR-load address field, and VM-entry MSR-load address field in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C).

The X2APIC MSRs cannot to be loaded and stored on VMX transitions. A VMX transition fails if the VMM has specified that the transition should access any MSRs in the address range from 0000_0800H to 0000_08FFH (the range used for accessing the X2APIC registers). Specifically, processing of an 128-bit entry in any of the VMX-transition MSR areas fails if bits 31:0 of that entry (represented as ENTRY_LOW_DW) satisfies the expression: "ENTRY_LOW_DW & FFFFF800H = 00000800H". Such a failure causes an associated VM entry to fail (by reloading host state) and causes an associated VM exit to lead to VMX abort.

12.12.5 x2APIC State Transitions

This section provides a detailed description of the x2APIC states of a local x2APIC unit, transitions between these states as well as interactions of these states with INIT and reset.

12.12.5.1 x2APIC States

The valid states for a local x2APIC unit are listed in Table 12-5.

- APIC disabled: IA32_APIC_BASE[EN]=0 and IA32_APIC_BASE[EXTD]=0.
- xAPIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=0.
- x2APIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=1.
- Invalid: IA32_APIC_BASE[EN]=0 and IA32_APIC_BASE[EXTD]=1.

The state corresponding to EXTD=1 and EN=0 is not valid and it is not possible to get into this state. An execution of WRMSR to the IA32_APIC_BASE_MSR that attempts a transition from a valid state to this invalid state causes a general-protection exception. Figure 12-27 shows the comprehensive state transition diagram for a local x2APIC unit.

On coming out of reset, the local APIC unit is enabled and is in the xAPIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=0. The APIC registers are initialized as follows.

- The local APIC ID is initialized by hardware with a 32 bit ID (x2APIC ID). The lowest 8 bits of the x2APIC ID are the legacy local xAPIC ID, and are stored in the upper 8 bits of the APIC register for access in xAPIC mode.
- The following APIC registers are reset to all zeros for those fields that are defined in the xAPIC mode.
  - IRR, ISR, TMR, ICR, LDR, TPR, Divide Configuration Register (See Section 12.4 through Section 12.6 for details of individual APIC registers).
  - Timer initial count and timer current count registers.
- The LVT registers are reset to 0s except for the mask bits; these are set to 1s.
- The local APIC version register is not affected.
- The Spurious Interrupt Vector Register is initialized to 000000FFH.
- The DFR (available only in xAPIC mode) is reset to all 1s.
- SELF IPI register is reset to zero.

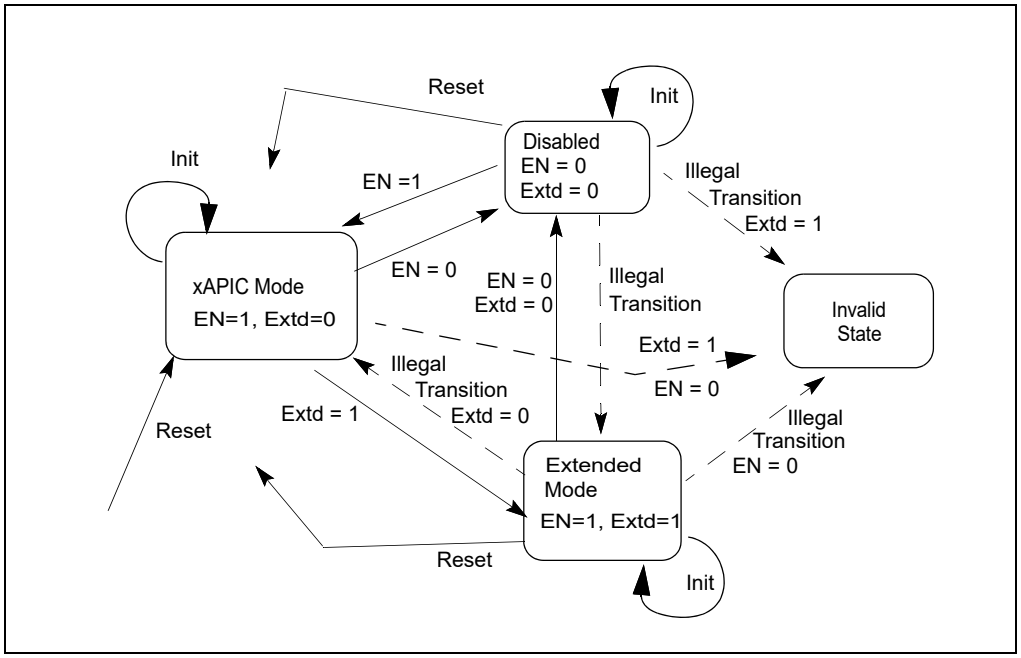


Figure 12-27. Local x2APIC State Transitions with IA32_APIC_BASE, INIT, and Reset

x2APIC After Reset

The valid transitions from the xAPIC mode state are:

- to the x2APIC mode by setting EXT to 1 (resulting EN=1, EXTD= 1). The physical x2APIC ID (see Figure 12-6) is preserved across this transition and the logical x2APIC ID (see Figure 12-29) is initialized by hardware during this transition as documented in Section 12.12.10.2. The state of the extended fields in other APIC registers, which was not initialized at reset, is not architecturally defined across this transition and system software should explicitly initialize those programmable APIC registers.
- to the disabled state by setting EN to 0 (resulting EN=0, EXTD= 0).

The result of an INIT in the xAPIC state places the APIC in the state with EN= 1, EXTD= 0. The state of the local APIC ID register is preserved (the 8-bit xAPIC ID is in the upper 8 bits of the APIC ID register). All the other APIC registers are initialized as a result of INIT.

A reset in this state places the APIC in the state with EN= 1, EXTD= 0. The state of the local APIC ID register is initialized as described in Section 12.12.5.1. All the other APIC registers are initialized described in Section 12.12.5.1.

x2APIC Transitions From x2APIC Mode

From the x2APIC mode, the only valid x2APIC transition using IA32_APIC_BASE is to the state where the x2APIC is disabled by setting EN to 0 and EXTD to 0. The x2APIC ID (32 bits) and the legacy local xAPIC ID (8 bits) are preserved across this transition. A transition from the x2APIC mode to xAPIC mode is not valid, and the corresponding WRMSR to the IA32_APIC_BASE MSR causes a general-protection exception.

A reset in this state places the x2APIC in xAPIC mode. All APIC registers (including the local APIC ID register) are initialized as described in Section 12.12.5.1.

An INIT in this state keeps the x2APIC in the x2APIC mode. The state of the local APIC ID register is preserved (all 32 bits). However, all the other APIC registers are initialized as a result of the INIT transition.

x2APIC Transitions From Disabled Mode

From the disabled state, the only valid x2APIC transition using IA32_APIC_BASE is to the xAPIC mode (EN= 1, EXTD = 0). Thus the only means to transition from x2APIC mode to xAPIC mode is a two-step process:

- first transition from x2APIC mode to local APIC disabled mode (EN= 0, EXTD = 0),
- followed by another transition from disabled mode to xAPIC mode (EN= 1, EXTD= 0).

Consequently, all the APIC register states in the x2APIC, except for the x2APIC ID (32 bits), are not preserved across mode transitions.

A reset in the disabled state places the x2APIC in the xAPIC mode. All APIC registers (including the local APIC ID register) are initialized as described in Section 12.12.5.1.

An INIT in the disabled state keeps the x2APIC in the disabled state.

State Changes From xAPIC Mode to x2APIC Mode

After APIC register states have been initialized by software in xAPIC mode, a transition from xAPIC mode to x2APIC mode does not affect most of the APIC register states, except the following:

- The Logical Destination Register is not preserved.
- Any APIC ID value written to the memory-mapped local APIC ID register is not preserved.
- The high half of the Interrupt Command Register is not preserved.

12.12.6 Routing of Device Interrupts in x2APIC Mode

The x2APIC architecture is intended to work with all existing IOxAPIC units as well as all PCI and PCI Express (PCIe) devices that support the capability for message-signaled interrupts (MSI). Support for x2APIC modifies only the following:

- the local APIC units;
- the interconnects joining IOxAPIC units to the local APIC units; and
- the interconnects joining MSI-capable PCI and PCIe devices to the local APIC units.

No modifications are required to MSI-capable PCI and PCIe devices. Similarly, no modifications are required to IOxAPIC units. This is made possible through use of the interrupt-remapping architecture specified in the Intel® Virtualization Technology for Directed I/O Specification, Revision 1.3 and/or later versions, for the routing of interrupts from MSI-capable devices to local APIC units operating in x2APIC mode.

12.12.7 Initialization by System Software

Routing of device interrupts to local APIC units operating in x2APIC mode requires use of the interrupt-remapping architecture specified in the Intel® Virtualization Technology for Directed I/O Specification (Revision 1.3 and/or later versions). Because of this, BIOS must enumerate support for and software must enable this interrupt remapping with Extended Interrupt Mode Enabled before it enabling x2APIC mode in the local APIC units.

The ACPI interfaces for the x2APIC are described in Section 5.2, "ACPI System Description Tables," of the Advanced Configuration and Power Interface Specification, Revision 4.0a (http://www.acpi.info/spec.htm). The default behavior for BIOS is to pass the control to the operating system with the local x2APICs in xAPIC mode if all APIC IDs reported by CPUID.0BH:EDX are less than 255, and in x2APIC mode if there are any logical processor reporting an APIC ID of 255 or greater.

12.12.8 CPUID Extensions And Topology Enumeration

For Intel 64 and IA-32 processors that support x2APIC, a value of 1 reported by CPUID.01H:ECX[21] indicates that the processor supports x2APIC and the extended topology enumeration leaf (CPUID.0BH).

The extended topology enumeration leaf can be accessed by executing CPUID with EAX = 0BH. Processors that do not support x2APIC may support CPUID leaf 0BH. Software can detect the availability of the extended topology enumeration leaf (0BH) by performing two steps:

- Check maximum input value for basic CPUID information by executing CPUID with EAX= 0. If CPUID.0H:EAX is greater than or equal or 11 (0BH), then proceed to next step
- Check CPUID.EAX=0BH, ECX=0H:EBX is non-zero.

If both of the above conditions are true, extended topology enumeration leaf is available. If available, the extended topology enumeration leaf is the preferred mechanism for enumerating topology. The presence of CPUID leaf 0BH in a processor does not guarantee support for x2APIC. If CPUID.EAX=0BH, ECX=0H:EBX returns zero and maximum input value for basic CPUID information is greater than 0BH, then CPUID.0BH leaf is not supported on that processor.

The extended topology enumeration leaf is intended to assist software with enumerating processor topology on systems that requires 32-bit x2APIC IDs to address individual logical processors. Details of CPUID leaf 0BH can be found in the reference pages of CPUID in Chapter 3 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

Processor topology enumeration algorithm for processors supporting the extended topology enumeration leaf of CPUID and processors that do not support CPUID leaf 0BH are treated in Section 10.9.4, "Algorithm for Three-Domain Mappings of APIC_ID."

12.12.8.1 Consistency of APIC IDs and CPUID

The consistency of physical x2APIC ID in MSR 802H in x2APIC mode and the 32-bit value returned in CPUID.0BH:EDX is facilitated by processor hardware.

CPUID.0BH:EDX will report the full 32 bit ID, in xAPIC and x2APIC mode. This allows BIOS to determine if a system has processors with IDs exceeding the 8-bit initial APIC ID limit (CPUID.01H:EBX[31:24]). Initial APIC ID (CPUID.01H:EBX[31:24]) is always equal to CPUID.0BH:EDX[7:0].

If the values of CPUID.0BH:EDX reported by all logical processors in a system are less than 255, BIOS can transfer control to OS in xAPIC mode.

If the values of CPUID.0BH:EDX reported by some logical processors in a system are greater than or equal to 255, BIOS must support two options to hand off to OS.

- If BIOS enables logical processors with x2APIC IDs greater than 255, then it should enable x2APIC in the Boot Strap Processor (BSP) and all Application Processors (AP) before passing control to the OS. Applications requiring processor topology information must use OS provided services based on x2APIC IDs or CPUID.0BH leaf.
- If a BIOS transfers control to OS in xAPIC mode, then the BIOS must ensure that only logical processors with CPUID.0BH.EDX value less than 255 are enabled. BIOS initialization on all logical processors with CPUID.0B.EDX values greater than or equal to 255 must (a) disable APIC and execute CLI in each logical processor, and (b) leave these logical processor in the lowest power state so that these processors do not respond to INIT IPI during OS boot. The BSP and all the enabled logical processor operate in xAPIC mode after BIOS passed control to OS. Application requiring processor topology information can use OS provided legacy services based on 8-bit initial APIC IDs or legacy topology information from CPUID.01H and CPUID 04H leaves. Even if the BIOS passes control in xAPIC mode, an OS can switch the processors to x2APIC mode later. BIOS SMM handler should always read the APIC_BASE_MSR, determine the APIC mode and use the corresponding access method.

12.12.9 ICR Operation in x2APIC Mode

In x2APIC mode, the layout of the Interrupt Command Register is shown in Figure 12-28. The lower 32 bits of ICR in x2APIC mode is identical to the lower half of the ICR in xAPIC mode, except the Delivery Status bit is removed since it is not needed in x2APIC mode. The destination ID field is expanded to 32 bits in x2APIC mode.

To send an IPI using the ICR, software must set up the ICR to indicate the type of IPI message to be sent and the destination processor or processors. Self IPIs can also be sent using the SELF IPI register (see Section 12.12.11).

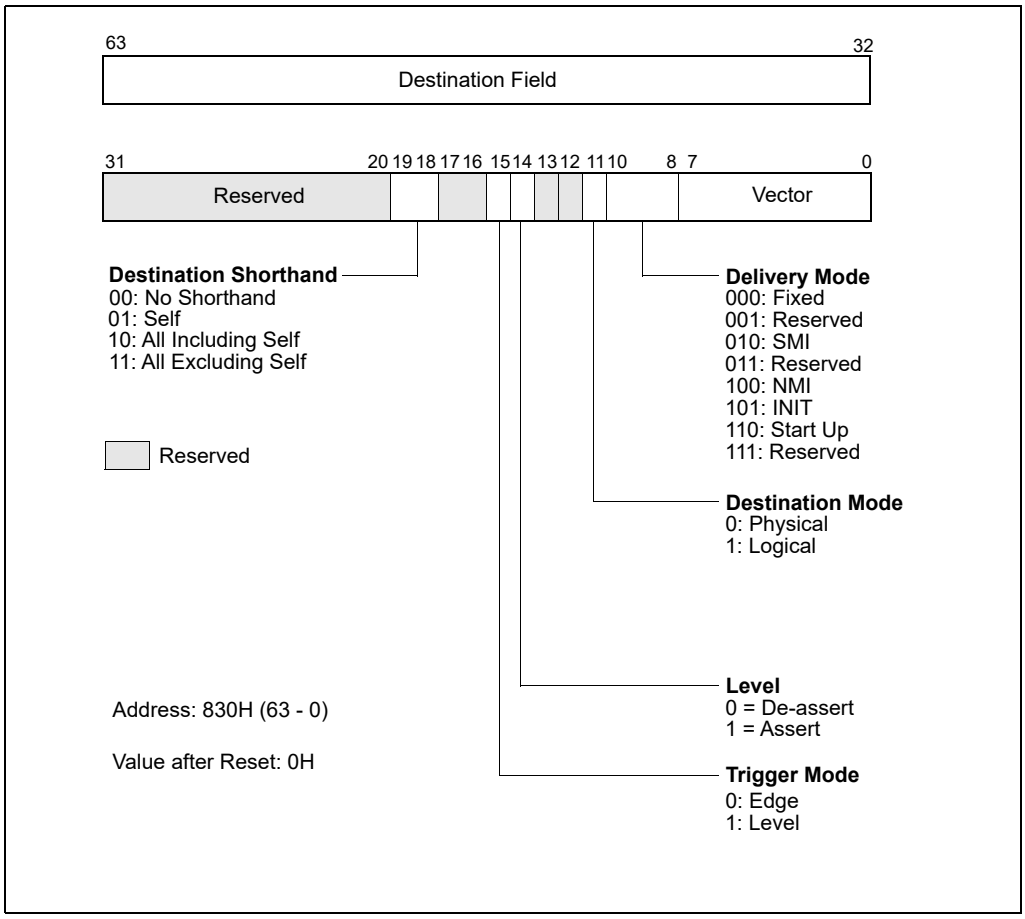


Figure 12-28. Interrupt Command Register (ICR) in x2APIC Mode

A single MSR write to the Interrupt Command Register is required for dispatching an interrupt in x2APIC mode. With the removal of the Delivery Status bit, system software no longer has a reason to read the ICR. It remains readable only to aid in debugging; however, software should not assume the value returned by reading the ICR is the last written value.

A destination ID value of FFFF_FFFFH is used for broadcast of interrupts in both logical destination and physical destination modes.

12.12.10 Determining IPI Destination in x2APIC Mode

12.12.10.1 Logical Destination Mode in x2APIC Mode

In x2APIC mode, the Logical Destination Register (LDR) is increased to 32 bits wide. It is a read-only register to system software. This 32-bit value is referred to as "logical x2APIC ID". System software accesses this register via the RDMSR instruction reading the MSR at address 80DH. Figure 12-29 provides the layout of the Logical Destination Register in x2APIC mode.

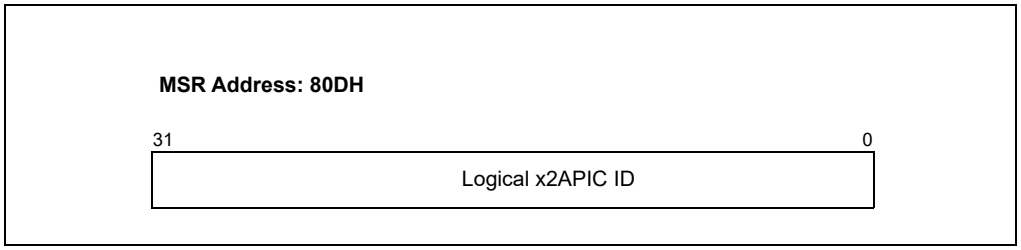


Figure 12-29. Logical Destination Register in x2APIC Mode

In the xAPIC mode, the Destination Format Register (DFR) through the MMIO interface determines the choice of a flat logical mode or a clustered logical mode. Flat logical mode is not supported in the x2APIC mode. Hence the Destination Format Register (DFR) is eliminated in x2APIC mode.

The 32-bit logical x2APIC ID field of LDR is partitioned into two sub-fields:

- Cluster ID (LDR[31:16]): is the address of the destination cluster
- Logical ID (LDR[15:0]): defines a logical ID of the individual local x2APIC within the cluster specified by LDR[31:16].

This layout enables 2^16-1 clusters each with up to 16 unique logical IDs - effectively providing an addressability of ((2^20) - 16) processors in logical destination mode.

It is likely that processor implementations may choose to support less than 16 bits of the cluster ID or less than 16-bits of the Logical ID in the Logical Destination Register. However system software should be agnostic to the number of bits implemented in the cluster ID and logical ID sub-fields. The x2APIC hardware initialization will ensure that the appropriately initialized logical x2APIC IDs are available to system software and reads of non-implemented bits return zero. This is a read-only register that software must read to determine the logical x2APIC ID of the processor. Specifically, software can apply a 16-bit mask to the lowest 16 bits of the logical x2APIC ID to identify the logical address of a processor within a cluster without needing to know the number of implemented bits in cluster ID and Logical ID sub-fields. Similarly, software can create a message destination address for cluster model, by bit-Oring the Logical X2APIC ID (31:0) of processors that have matching Cluster ID(31:16).

To enable cluster ID assignment in a fashion that matches the system topology characteristics and to enable efficient routing of logical mode lowest priority device interrupts in link based platform interconnects, the LDR are initialized by hardware based on the value of x2APIC ID upon x2APIC state transitions. Details of this initialization are provided in Section 12.12.10.2.

12.12.10.2 Deriving Logical x2APIC ID from the Local x2APIC ID

In x2APIC mode, the 32-bit logical x2APIC ID, which can be read from LDR, is derived from the 32-bit local x2APIC ID. Specifically, the 16-bit logical ID sub-field is derived by shifting 1 by the lowest 4 bits of the x2APIC ID, i.e., Logical ID = 1 « x2APIC ID[3:0]. The remaining bits of the x2APIC ID then form the cluster ID portion of the logical x2APIC ID:

```
Logical x2APIC ID = [(x2APIC ID[19:4] « 16) | (1 « x2APIC ID[3:0])]
```

The use of the lowest 4 bits in the x2APIC ID implies that at least 16 APIC IDs are reserved for logical processors within a socket in multi-socket configurations. If more than 16 APIC IDS are reserved for logical processors in a socket/package then multiple cluster IDs can exist within the package.

The LDR initialization occurs whenever the x2APIC mode is enabled (see Section 12.12.5).

12.12.11 SELF IPI Register

SELF IPIs are used extensively by some system software. The x2APIC architecture introduces a new register interface. This new register is dedicated to the purpose of sending self-IPIs with the intent of enabling a highly optimized path for sending self-IPIs.

Figure 12-30 provides the layout of the SELF IPI register. System software only specifies the vector associated with the interrupt to be sent. The semantics of sending a self-IPI via the SELF IPI register are identical to sending a self targeted edge triggered fixed interrupt with the specified vector. Specifically the semantics are identical to the following settings for an inter-processor interrupt sent via the ICR - Destination Shorthand (ICR[19:18] = 01 (Self)), Trigger Mode (ICR[15] = 0 (Edge)), Delivery Mode (ICR[10:8] = 000 (Fixed)), Vector (ICR[7:0] = Vector).

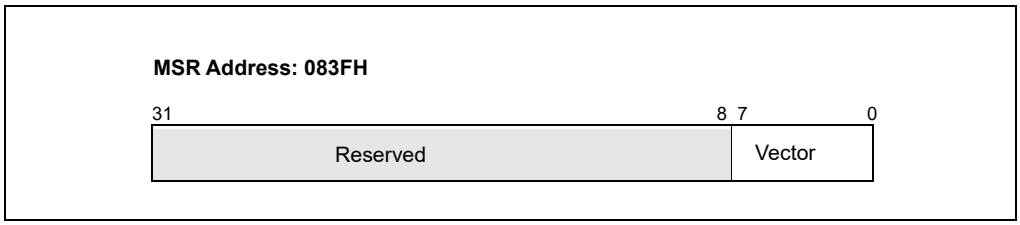


Figure 12-30. SELF IPI register

The SELF IPI register is a write-only register. A RDMSR instruction with address of the SELF IPI register causes a general-protection exception.

The handling and prioritization of a self-IPI sent via the SELF IPI register is architecturally identical to that for an IPI sent via the ICR from a legacy xAPIC unit. Specifically the state of the interrupt would be tracked via the Interrupt Request Register (IRR) and In Service Register (ISR) and Trigger Mode Register (TMR) as if it were received from the system bus. Also sending the IPI via the Self Interrupt Register ensures that interrupt is delivered to the processor core. Specifically completion of the WRMSR instruction to the SELF IPI register implies that the interrupt has been logged into the IRR. As expected for edge triggered interrupts, depending on the processor priority and readiness to accept interrupts, it is possible that interrupts sent via the SELF IPI register or via the ICR with identical vectors can be combined.

12.13 APIC BUS MESSAGE FORMATS

This section describes the message formats used when transmitting messages on the serial APIC bus. The information described here pertains only to the Pentium and P6 family processors.

12.13.1 Bus Message Formats

The local and I/O APICs transmit three types of messages on the serial APIC bus: EOI message, short message, and non-focused lowest priority message. The purpose of each type of message and its format are described below.

12.13.2 EOI Message

Local APICs send 14-cycle EOI messages to the I/O APIC to indicate that a level triggered interrupt has been accepted by the processor. This interrupt, in turn, is a result of software writing into the EOI register of the local APIC. Table 12-8 shows the cycles in an EOI message.

Table 12-8. EOI Message (14 Cycles)

Cycle	Bit1	Bit0	
1	1	1	11 = EOI
2	ArbID3	0	Arbitration ID bits 3 through 0

Table 12-8. EOI Message (14 Cycles) (Contd.)

Cycle	Bit1	Bit0	
3	ArbID2	0	
4	ArbID1	0	
5	ArbID0	0	
6	V7	V6	Interrupt vector V7 - V0
7	V5	V4	
8	V3	V2	
9	V1	V0	
10	C	C	Checksum for cycles 6 - 9
11	0	0	
12	A	A	Status Cycle 0
13	A1	A1	Status Cycle 1
14	0	0	Idle

The checksum is computed for cycles 6 through 9. It is a cumulative sum of the 2-bit (Bit1:Bit0) logical data values. The carry out of all but the last addition is added to the sum. If any APIC computes a different checksum than the one appearing on the bus in cycle 10, it signals an error, driving 11 on the APIC bus during cycle 12. In this case, the APICs disregard the message. The sending APIC will receive an appropriate error indication (see Section 12.5.3, "Error Handling") and resend the message. The status cycles are defined in Table 12-11.

12.13.2.1 Short Message

Short messages (21-cycles) are used for sending fixed, NMI, SMI, INIT, start-up, ExtINT, and lowest-priority-with-focus interrupts. Table 12-9 shows the cycles in a short message.

Table 12-9. Short Message (21 Cycles)

Cycle	Bit1	Bit0	
1	0	1	0 1 = normal
2	ArbID3	0	Arbitration ID bits 3 through 0
3	ArbID2	0	
4	ArbID1	0	
5	ArbID0	0	
6	DM	M2	DM = Destination Mode
7	M1	M0	M2-M0 = Delivery mode
8	L	TM	L = Level, TM = Trigger Mode
9	V7	V6	V7-V0 = Interrupt Vector
10	V5	V4	
11	V3	V2	
12	V1	V0	
13	D7	D6	D7-D0 = Destination
14	D5	D4	
15	D3	D2	
16	D1	D0	

Table 12-9. Short Message (21 Cycles) (Contd.)

Cycle	Bit1	Bit0	
17	C	C	Checksum for cycles 6-16
18	0	0	
19	A	A	Status cycle 0
20	A1	A1	Status cycle 1
21	0	0	Idle

If the physical delivery mode is being used, then cycles 15 and 16 represent the APIC ID and cycles 13 and 14 are considered don't care by the receiver. If the logical delivery mode is being used, then cycles 13 through 16 are the 8-bit logical destination field.

For shorthands of "all-incl-self" and "all-excl-self," the physical delivery mode and an arbitration priority of 15 (D0:D3 = 1111) are used. The agent sending the message is the only one required to distinguish between the two cases. It does so using internal information.

When using lowest priority delivery with an existing focus processor, the focus processor identifies itself by driving 10 during cycle 19 and accepts the interrupt. This is an indication to other APICs to terminate arbitration. If the focus processor has not been found, the short message is extended on-the-fly to the non-focused lowest-priority message. Note that except for the EOI message, messages generating a checksum or an acceptance error (see Section 12.5.3, "Error Handling") terminate after cycle 21.

12.13.2.2 Non-focused Lowest Priority Message

These 34-cycle messages (see Table 12-10) are used in the lowest priority delivery mode when a focus processor is not present. Cycles 1 through 20 are same as for the short message. If during the status cycle (cycle 19) the state of the (A:A) flags is 10B, a focus processor has been identified, and the short message format is used (see Table 12-9). If the (A:A) flags are set to 00B, lowest priority arbitration is started and the 34-cycles of the non-focused lowest priority message are competed. For other combinations of status flags, refer to Section 12.13.2.3, "APIC Bus Status Cycles."

Table 12-10. Non-Focused Lowest Priority Message (34 Cycles)

Cycle	Bit0	Bit1	
1	0	1	0 1 = normal
2	ArbID3	0	Arbitration ID bits 3 through 0
3	ArbID2	0	
4	ArbID1	0	
5	ArbID0	0	
6	DM	M2	DM = Destination mode
7	M1	M0	M2-M0 = Delivery mode
8	L	TM	L = Level, TM = Trigger Mode
9	V7	V6	V7-V0 = Interrupt Vector
10	V5	V4	
11	V3	V2	
12	V1	V0	
13	D7	D6	D7-D0 = Destination
14	D5	D4	
15	D3	D2	
16	D1	D0	

Table 12-10. Non-Focused Lowest Priority Message (34 Cycles) (Contd.)

Cycle	Bit0	Bit1	
17	C	C	Checksum for cycles 6-16
18	0	0	
19	A	A	Status cycle 0
20	A1	A1	Status cycle 1
21	P7	0	P7 - P0 = Inverted Processor Priority
22	P6	0	
23	P5	0	
24	P4	0	
25	P3	0	
26	P2	0	
27	P1	0	
28	P0	0	
29	ArbID3	0	Arbitration ID 3 -0
30	ArbID2	0	
31	ArbID1	0	
32	ArbID0	0	
33	A2	A2	Status Cycle
34	0	0	Idle

Cycles 21 through 28 are used to arbitrate for the lowest priority processor. The processors participating in the arbitration drive their inverted processor priority on the bus. Only the local APICs having free interrupt slots participate in the lowest priority arbitration. If no such APIC exists, the message will be rejected, requiring it to be tried at a later time.

Cycles 29 through 32 are also used for arbitration in case two or more processors have the same lowest priority. In the lowest priority delivery mode, all combinations of errors in cycle 33 (A2 A2) will set the "accept error" bit in the error status register (see Figure 12-9). Arbitration priority update is performed in cycle 20, and is not affected by errors detected in cycle 33. Only the local APIC that wins in the lowest priority arbitration, drives cycle 33. An error in cycle 33 will force the sender to resend the message.

12.13.2.3 APIC Bus Status Cycles

Certain cycles within an APIC bus message are status cycles. During these cycles the status flags (A:A) and (A1:A1) are examined. Table 12-11 shows how these status flags are interpreted, depending on the current delivery mode and existence of a focus processor.

Table 12-11. APIC Bus Status Cycles Interpretation

Delivery Mode	A Status	A1 Status	A2 Status	Update ArbID and Cycle#	Message Length	Retry
EOI	00: CS_OK	10: Accept	XX:	Yes, 13	14 Cycle	No
	00: CS_OK	11: Retry	XX:	Yes, 13	14 Cycle	Yes
	00: CS_OK	0X: Accept Error	XX:	No	14 Cycle	Yes
	11: CS_Error	XX:	XX:	No	14 Cycle	Yes
	10: Error	XX:	XX:	No	14 Cycle	Yes
	01: Error	XX:	XX:	No	14 Cycle	Yes
Fixed	00: CS_OK	10: Accept	XX:	Yes, 20	21 Cycle	No
	00: CS_OK	11: Retry	XX:	Yes, 20	21 Cycle	Yes
	00: CS_OK	0X: Accept Error	XX:	No	21 Cycle	Yes
	11: CS_Error	XX:	XX:	No	21 Cycle	Yes
	10: Error	XX:	XX:	No	21 Cycle	Yes
	01: Error	XX:	XX:	No	21 Cycle	Yes
NMI, SMI, INIT, ExtINT, Start-Up	00: CS_OK	10: Accept	XX:	Yes, 20	21 Cycle	No
	00: CS_OK	11: Retry	XX:	Yes, 20	21 Cycle	Yes
	00: CS_OK	0X: Accept Error	XX:	No	21 Cycle	Yes
	11: CS_Error	XX:	XX:	No	21 Cycle	Yes
	10: Error	XX:	XX:	No	21 Cycle	Yes
	01: Error	XX:	XX:	No	21 Cycle	Yes
Lowest	00: CS_OK, NoFocus	11: Do Lowest	10: Accept	Yes, 20	34 Cycle	No
	00: CS_OK, NoFocus	11: Do Lowest	11: Error	Yes, 20	34 Cycle	Yes
	00: CS_OK, NoFocus	11: Do Lowest	0X: Error	Yes, 20	34 Cycle	Yes
	00: CS_OK, NoFocus	10: End and Retry	XX:	Yes, 20	34 Cycle	Yes
	00: CS_OK, NoFocus	0X: Error	XX:	No	34 Cycle	Yes
	10: CS_OK, Focus	XX:	XX:	Yes, 20	34 Cycle	No
	11: CS_Error	XX:	XX:	No	21 Cycle	Yes
	01: Error	XX:	XX:	No	21 Cycle	Yes

CHAPTER 13
MEMORY CACHE CONTROL

This chapter describes the memory cache and cache control mechanisms, the TLBs, and the store buffer in Intel 64 and IA-32 processors. It also describes the memory type range registers (MTRRs) introduced in the P6 family processors and how they are used to control caching of physical memory locations.

13.1 INTERNAL CACHES, TLBS, AND BUFFERS

The Intel 64 and IA-32 architectures support cache, translation look aside buffers (TLBs), and a store buffer for temporary on-chip (and external) storage of instructions and data. (Figure 13-1 shows the arrangement of caches, TLBs, and the store buffer for the Pentium 4 and Intel Xeon processors.) Table 13-1 shows the characteristics of these caches and buffers for the Pentium 4, Intel Xeon, P6 family, and Pentium processors. **The sizes and characteristics of these units are machine specific and may change in future versions of the processor.** The CPUID instruction returns the sizes and characteristics of the caches and buffers for the processor on which the instruction is executed. See "CPUID—CPU Identification" in Chapter 3, "Instruction Set Reference, A-L," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.

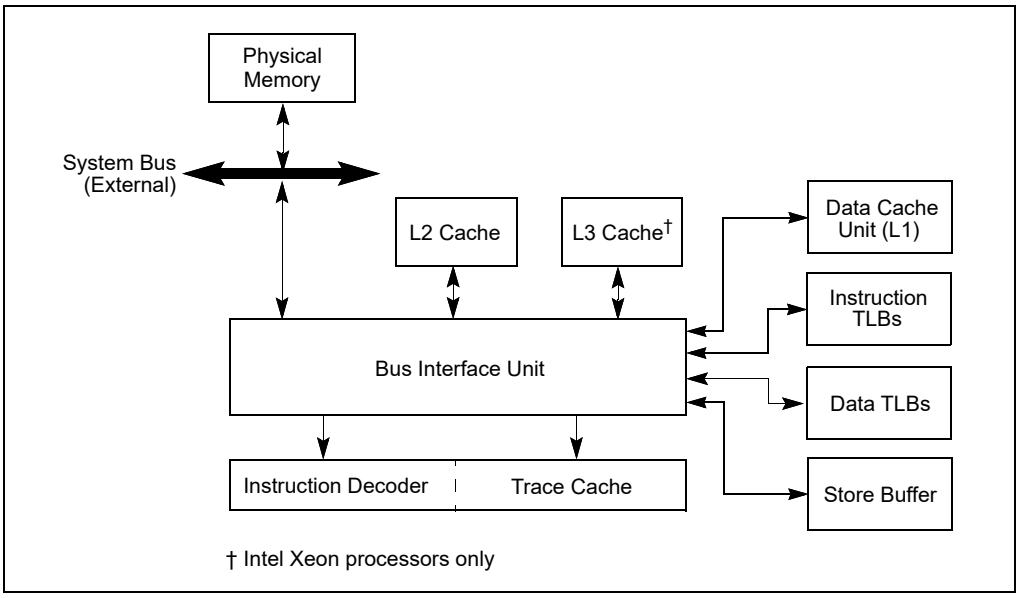


Figure 13-1. Cache Structure of the Pentium 4 and Intel Xeon Processors

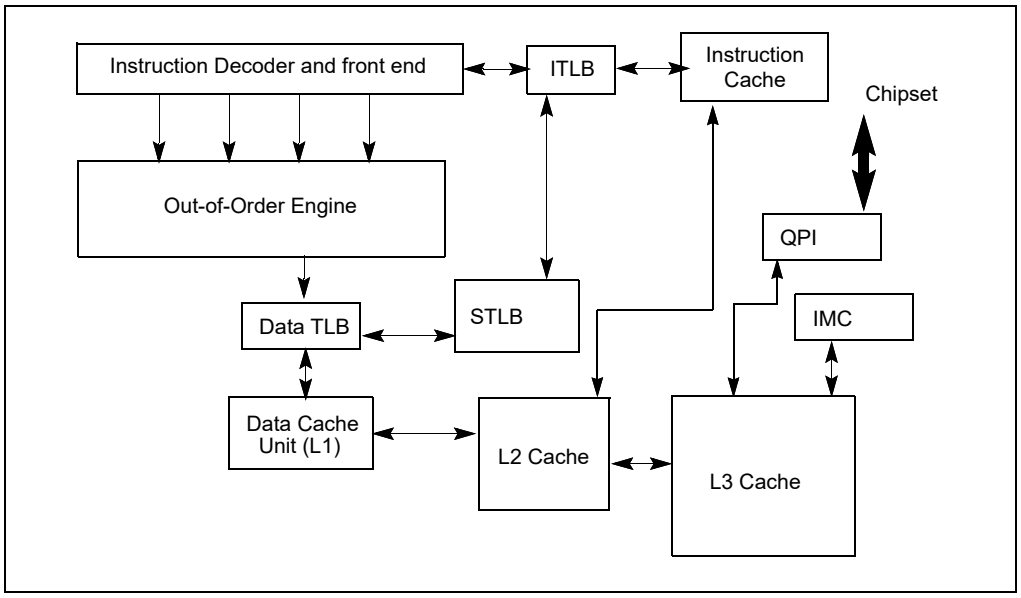


Figure 13-2. Cache Structure of the Intel Core i7 Processors

Figure 13-2 shows the cache arrangement of Intel Core i7 processor.

Table 13-1. Characteristics of the Caches, TLBs, Store Buffer, and Write Combining Buffer in Intel 64 and IA-32 Processors

Cache or Buffer	Characteristics
Trace Cache[1]	▪ Pentium 4 and Intel Xeon processors (Based on Intel NetBurst® microarchitecture): 12 Kμops, 8-way set associative. ▪ Intel Core i7, Intel Core 2 Duo, Intel Atom, Intel Core Duo, Intel Core Solo, Pentium M processor: not implemented. ▪ P6 family and Pentium processors: not implemented.
L1 Instruction Cache	▪ Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): not implemented. ▪ Intel Core i7 processor: 32-KByte, 4-way set associative. ▪ Intel Core 2 Duo, Intel Atom, Intel Core Duo, Intel Core Solo, Pentium M processor: 32-KByte, 8-way set associative. ▪ P6 family and Pentium processors: 8- or 16-KByte, 4-way set associative, 32-byte cache line size; 2-way set associative for earlier Pentium processors.
L1 Data Cache	▪ Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): 8-KByte, 4-way set associative, 64-byte cache line size. ▪ Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): 16-KByte, 8-way set associative, 64-byte cache line size. ▪ Intel Atom processors: 24-KByte, 6-way set associative, 64-byte cache line size. ▪ Intel Core i7, Intel Core 2 Duo, Intel Core Duo, Intel Core Solo, Pentium M and Intel Xeon processors: 32-KByte, 8-way set associative, 64-byte cache line size. ▪ P6 family processors: 16-KByte, 4-way set associative, 32-byte cache line size; 8-KBytes, 2-way set associative for earlier P6 family processors. ▪ Pentium processors: 16-KByte, 4-way set associative, 32-byte cache line size; 8-KByte, 2-way set associative for earlier Pentium processors.

Table 13-1. Characteristics of the Caches, TLBs, Store Buffer, and Write Combining Buffer in Intel 64 and IA-32 Processors (Contd.)

Cache or Buffer	Characteristics
L2 Unified Cache	▪ Intel Core 2 Duo and Intel Xeon processors: up to 4-MByte (or 4MBx2 in quadcore processors), 16-way set associative, 64-byte cache line size. ▪ Intel Core 2 Duo and Intel Xeon processors: up to 6-MByte (or 6MBx2 in quadcore processors), 24-way set associative, 64-byte cache line size. ▪ Intel Core i7, i5, i3 processors: 256KBbyte, 8-way set associative, 64-byte cache line size. ▪ Intel Atom processors: 512-KByte, 8-way set associative, 64-byte cache line size. ▪ Intel Core Duo, Intel Core Solo processors: 2-MByte, 8-way set associative, 64-byte cache line size ▪ Pentium 4 and Intel Xeon processors: 256, 512, 1024, or 2048-KByte, 8-way set associative, 64-byte cache line size, 128-byte sector size. ▪ Pentium M processor: 1 or 2-MByte, 8-way set associative, 64-byte cache line size. ▪ P6 family processors: 128-KByte, 256-KByte, 512-KByte, 1-MByte, or 2-MByte, 4-way set associative, 32-byte cache line size. ▪ Pentium processor (external optional): System specific, typically 256- or 512-KByte, 4-way set associative, 32-byte cache line size.
L3 Unified Cache	▪ Intel Xeon processors: 512-KByte, 1-MByte, 2-MByte, or 4-MByte, 8-way set associative, 64-byte cache line size, 128-byte sector size. ▪ Intel Core i7 processor, Intel Xeon processor 5500: Up to 8MByte, 16-way set associative, 64-byte cache line size. ▪ Intel Xeon processor 5600: Up to 12MByte, 64-byte cache line size. ▪ Intel Xeon processor 7500: Up to 24MByte, 64-byte cache line size.
Instruction TLB (4-KByte Pages)	▪ Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): 128 entries, 4-way set associative. ▪ Intel Atom processors: 32-entries, fully associative. ▪ Intel Core i7, i5, i3 processors: 64-entries per thread (128-entries per core), 4-way set associative. ▪ Intel Core 2 Duo, Intel Core Duo, Intel Core Solo processors, Pentium M processor: 128 entries, 4-way set associative. ▪ P6 family processors: 32 entries, 4-way set associative. ▪ Pentium processor: 32 entries, 4-way set associative; fully set associative for Pentium processors with MMX technology.
Data TLB (4-KByte Pages)	▪ Intel Core i7, i5, i3 processors, DTLB0: 64-entries, 4-way set associative. ▪ Intel Core 2 Duo processors: DTLB0, 16 entries, DTLB1, 256 entries, 4 ways. ▪ Intel Atom processors: 16-entry-per-thread micro-TLB, fully associative; 64-entry DTLB, 4-way set associative; 16-entry PDE cache, fully associative. ▪ Pentium 4 and Intel Xeon processors (Based on Intel NetBurst microarchitecture): 64 entry, fully set associative, shared with large page DTLB. ▪ Intel Core Duo, Intel Core Solo processors, Pentium M processor: 128 entries, 4-way set associative. ▪ Pentium and P6 family processors: 64 entries, 4-way set associative; fully set, associative for Pentium processors with MMX technology.
Instruction TLB (Large Pages)	▪ Intel Core i7, i5, i3 processors: 7-entries per thread, fully associative. ▪ Intel Core 2 Duo processors: 4 entries, 4 ways. ▪ Pentium 4 and Intel Xeon processors: large pages are fragmented. ▪ Intel Core Duo, Intel Core Solo, Pentium M processor: 2 entries, fully associative. ▪ P6 family processors: 2 entries, fully associative. ▪ Pentium processor: Uses same TLB as used for 4-KByte pages.
Data TLB (Large Pages)	▪ Intel Core i7, i5, i3 processors, DTLB0: 32-entries, 4-way set associative. ▪ Intel Core 2 Duo processors: DTLB0, 16 entries, DTLB1, 32 entries, 4 ways. ▪ Intel Atom processors: 8 entries, 4-way set associative. ▪ Pentium 4 and Intel Xeon processors: 64 entries, fully set associative; shared with small page data TLBs. ▪ Intel Core Duo, Intel Core Solo, Pentium M processor: 8 entries, fully associative. ▪ P6 family processors: 8 entries, 4-way set associative. ▪ Pentium processor: 8 entries, 4-way set associative; uses same TLB as used for 4-KByte pages in Pentium processors with MMX technology.
Second-level Unified TLB (4-KByte Pages)	▪ Intel Core i7, i5, i3 processor, STLB: 512-entries, 4-way set associative.

Table 13-1. Characteristics of the Caches, TLBs, Store Buffer, and Write Combining Buffer in Intel 64 and IA-32 Processors (Contd.)

Cache or Buffer	Characteristics
Store Buffer	▪ Intel Core i7, i5, i3 processors: 32entries. ▪ Intel Core 2 Duo processors: 20 entries. ▪ Intel Atom processors: 8 entries, used for both WC and store buffers. ▪ Pentium 4 and Intel Xeon processors: 24 entries. ▪ Pentium M processor: 16 entries. ▪ P6 family processors: 12 entries. ▪ Pentium processor: 2 buffers, 1 entry each (Pentium processors with MMX technology have 4 buffers for 4 entries).
Write Combining (WC) Buffer	▪ Intel Core 2 Duo processors: 8 entries. ▪ Intel Atom processors: 8 entries, used for both WC and store buffers. ▪ Pentium 4 and Intel Xeon processors: 6 or 8 entries. ▪ Intel Core Duo, Intel Core Solo, Pentium M processors: 6 entries. ▪ P6 family processors: 4 entries.

NOTES:

1 Introduced to the IA-32 architecture in the Pentium 4 and Intel Xeon processors.

Intel 64 and IA-32 processors may implement four types of caches: the trace cache, the level 1 (L1) cache, the level 2 (L2) cache, and the level 3 (L3) cache. See Figure 13-1. Cache availability is described below:

- **Intel Core i7, i5, i3 processor family and Intel Xeon processor family based on Nehalem microarchitecture and Westmere microarchitecture** — The L1 cache is divided into two sections: one section is dedicated to caching instructions (pre-decoded instructions) and the other caches data. The L2 cache is a unified data and instruction cache. Each processor core has its own L1 and L2. The L3 cache is an inclusive, unified data and instruction cache, shared by all processor cores inside a physical package. No trace cache is implemented.
- **Intel® Core™ 2 processor family and Intel® Xeon® processor family based on Intel® Core™ microarchitecture** — The L1 cache is divided into two sections: one section is dedicated to caching instructions (pre-decoded instructions) and the other caches data. The L2 cache is a unified data and instruction cache located on the processor chip; it is shared between two processor cores in a dual-core processor implementation. Quad-core processors have two L2, each shared by two processor cores. No trace cache is implemented.
- **Intel Atom® processor** — The L1 cache is divided into two sections: one section is dedicated to caching instructions (pre-decoded instructions) and the other caches data. The L2 cache is a unified data and instruction cache is located on the processor chip. No trace cache is implemented.
- **Intel® Core™ Solo and Intel® Core™ Duo processors** — The L1 cache is divided into two sections: one section is dedicated to caching instructions (pre-decoded instructions) and the other caches data. The L2 cache is a unified data and instruction cache located on the processor chip. It is shared between two processor cores in a dual-core processor implementation. No trace cache is implemented.
- **Pentium® 4 and Intel® Xeon® processors Based on Intel NetBurst® microarchitecture** — The trace cache caches decoded instructions (μops) from the instruction decoder and the L1 cache contains data. The L2 and L3 caches are unified data and instruction caches located on the processor chip. Dualcore processors have two L2, one in each processor core. Note that the L3 cache is only implemented on some Intel Xeon processors.
- **P6 family processors** — The L1 cache is divided into two sections: one dedicated to caching instructions (pre-decoded instructions) and the other to caching data. The L2 cache is a unified data and instruction cache located on the processor chip. P6 family processors do not implement a trace cache.
- **Pentium® processors** — The L1 cache has the same structure as on P6 family processors. There is no trace cache. The L2 cache is a unified data and instruction cache external to the processor chip on earlier Pentium processors and implemented on the processor chip in later Pentium processors. For Pentium processors where the L2 cache is external to the processor, access to the cache is through the system bus.

For Intel Core i7 processors and processors based on Intel Core, Intel Atom, and Intel NetBurst microarchitectures, Intel Core Duo, Intel Core Solo and Pentium M processors, the cache lines for the L1 and L2 caches (and L3 caches if supported) are 64 bytes wide. The processor always reads a cache line from system memory beginning on a 64-byte boundary. (A 64-byte aligned cache line begins at an address with its 6 least-significant bits clear.) A cache

line can be filled from memory with a 8-transfer burst transaction. The caches do not support partially-filled cache lines, so caching even a single doubleword requires caching an entire line.

The L1 and L2 cache lines in the P6 family and Pentium processors are 32 bytes wide, with cache line reads from system memory beginning on a 32-byte boundary (5 least-significant bits of a memory address clear.) A cache line can be filled from memory with a 4-transfer burst transaction. Partially-filled cache lines are not supported.

The trace cache in processors based on Intel NetBurst microarchitecture is available in all execution modes: protected mode, system management mode (SMM), and real-address mode. The L1,L2, and L3 caches are also available in all execution modes; however, use of them must be handled carefully in SMM (see Section 33.4.2, "SMRAM Caching").

The TLBs store the most recently used page-directory and page-table entries. They speed up memory accesses when paging is enabled by reducing the number of memory accesses that are required to read the page tables stored in system memory. The TLBs are divided into four groups: instruction TLBs for 4-KByte pages, data TLBs for 4-KByte pages; instruction TLBs for large pages (2-MByte, 4-MByte or 1-GByte pages), and data TLBs for large pages. The TLBs are normally active only in protected mode with paging enabled. When paging is disabled or the processor is in real-address mode, the TLBs maintain their contents until explicitly or implicitly flushed (see Section 13.9, "Invalidating the Translation Lookaside Buffers (TLBs)").

Processors based on Intel Core microarchitectures implement one level of instruction TLB and two levels of data TLB. Intel Core i7 processor provides a second-level unified TLB.

The store buffer is associated with the processors instruction execution units. It allows writes to system memory and/or the internal caches to be saved and in some cases combined to optimize the processor's bus accesses. The store buffer is always enabled in all execution modes.

The processor's caches are for the most part transparent to software. When enabled, instructions and data flow through these caches without the need for explicit software control. However, knowledge of the behavior of these caches may be useful in optimizing software performance. For example, knowledge of cache dimensions and replacement algorithms gives an indication of how large of a data structure can be operated on at once without causing cache thrashing.

In multiprocessor systems, maintenance of cache consistency may, in rare circumstances, require intervention by system software. For these rare cases, the processor provides privileged cache control instructions for use in flushing caches and forcing memory ordering.

There are several instructions that software can use to improve the performance of the L1, L2, and L3 caches, including the PREFETCHh, CLFLUSH, and CLFLUSHOPT instructions and the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD). The use of these instructions are discussed in Section 13.5.5, "Cache Management Instructions."

13.2 CACHING TERMINOLOGY

IA-32 processors (beginning with the Pentium processor) and Intel 64 processors use the MESI (modified, exclusive, shared, invalid) cache protocol to maintain consistency with internal caches and caches in other processors (see Section 13.4, "Cache Control Protocol").

When the processor recognizes that an operand being read from memory is cacheable, the processor reads an entire cache line into the appropriate cache (L1, L2, L3, or all). This operation is called a **cache line fill**. If the memory location containing that operand is still cached the next time the processor attempts to access the operand, the processor can read the operand from the cache instead of going back to memory. This operation is called a **cache hit**.

When the processor attempts to write an operand to a cacheable area of memory, it first checks if a cache line for that memory location exists in the cache. If a valid cache line does exist, the processor (depending on the write policy currently in force) can write the operand into the cache instead of writing it out to system memory. This operation is called a **write hit**. If a write misses the cache (that is, a valid cache line is not present for area of memory being written to), the processor performs a cache line fill, write allocation. Then it writes the operand into the cache line and (depending on the write policy currently in force) can also write it out to memory. If the operand is to be written out to memory, it is written first into the store buffer, and then written from the store buffer to memory when the system bus is available. (Note that for the Pentium processor, write misses do not result in a cache line fill; they always result in a write to memory. For this processor, only read misses result in cache line fills.)

When operating in an MP system, IA-32 processors (beginning with the Intel486 processor) and Intel 64 processors have the ability to **snoop** other processor's accesses to system memory and to their internal caches. They use this snooping ability to keep their internal caches consistent both with system memory and with the caches in other processors on the bus. For example, in the Pentium and P6 family processors, if through snooping one processor detects that another processor intends to write to a memory location that it currently has cached in **shared state**, the snooping processor will invalidate its cache line forcing it to perform a cache line fill the next time it accesses the same memory location.

Beginning with the P6 family processors, if a processor detects (through snooping) that another processor is trying to access a memory location that it has modified in its cache, but has not yet written back to system memory, the snooping processor will signal the other processor (by means of the HITM# signal) that the cache line is held in modified state and will perform an implicit write-back of the modified data. The implicit write-back is transferred directly to the initial requesting processor and snooped by the memory controller to assure that system memory has been updated. Here, the processor with the valid data may pass the data to the other processors without actually writing it to system memory; however, it is the responsibility of the memory controller to snoop this operation and update memory.

13.3 METHODS OF CACHING AVAILABLE

The processor allows any area of system memory to be cached in the L1, L2, and L3 caches. In individual pages or regions of system memory, it allows the type of caching (also called **memory type**) to be specified (see Section 13.5). Memory types currently defined for the Intel 64 and IA-32 architectures are (see Table 13-2):

- **Strong Uncacheable (UC)** —System memory locations are not cached. All reads and writes appear on the system bus and are executed in program order without reordering. No speculative memory accesses, page-table walks, or prefetches of speculated branch targets are made. This type of cache-control is useful for memory-mapped I/O devices. When used with normal RAM, it greatly reduces processor performance.

NOTE

The behavior of x87 and SIMD instructions referencing memory is implementation dependent. In some implementations, accesses to UC memory may occur more than once. To ensure predictable behavior, use loads and stores of general purpose registers to access UC memory that may have read or write side effects.

Table 13-2. Memory Types and Their Properties

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Strong Uncacheable (UC)	No	No	No	Strong Ordering
Uncacheable (UC-)	No	No	No	Strong Ordering. Can only be selected through the PAT. Can be overridden by WC in MTRRs.
Write Combining (WC)	No	No	Yes	Weak Ordering. Available by programming MTRRs or by selecting it through the PAT.
Write Through (WT)	Yes	No	Yes	Speculative Processor Ordering.
Write Back (WB)	Yes	Yes	Yes	Speculative Processor Ordering.
Write Protected (WP)	Yes for reads; no for writes	No	Yes	Speculative Processor Ordering. Available by programming MTRRs.

- **Uncacheable (UC-)** — Has same characteristics as the strong uncacheable (UC) memory type, except that this memory type can be overridden by programming the MTRRs for the WC memory type. This memory type is available in processor families starting from the Pentium III processors and can only be selected through the PAT.

- **Write Combining (WC)** — System memory locations are not cached (as with uncacheable memory) and coherency is not enforced by the processor's bus coherency protocol. Speculative reads are allowed. Writes may be delayed and combined in the write combining buffer (WC buffer) to reduce memory accesses. If the WC buffer is partially filled, the writes may be delayed until the next occurrence of a serializing event; such as an SFENCE or MFENCE instruction, CPUID or other serializing instruction, a read or write to uncached memory, an interrupt occurrence, or an execution of a LOCK instruction (including one with an XACQUIRE or XRELEASE prefix). In addition, an execution of the XEND instruction (to end a transactional region) evicts any writes that were buffered before the corresponding execution of the XBEGIN instruction (to begin the transactional region) before evicting any writes that were performed inside the transactional region.

  This type of cache-control is appropriate for video frame buffers, where the order of writes is unimportant as long as the writes update memory so they can be seen on the graphics display. See Section 13.3.1, "Buffering of Write Combining Memory Locations," for more information about caching the WC memory type. This memory type is available in the Pentium Pro and Pentium II processors by programming the MTRRs; or in processor families starting from the Pentium III processors by programming the MTRRs or by selecting it through the PAT.
- **Write-through (WT)** — Writes and reads to and from system memory are cached. Reads come from cache lines on cache hits; read misses cause cache fills. Speculative reads are allowed. All writes are written to a cache line (when possible) and through to system memory. When writing through to memory, invalid cache lines are never filled, and valid cache lines are either filled or invalidated. Write combining is allowed. This type of cache-control is appropriate for frame buffers or when there are devices on the system bus that access system memory, but do not perform snooping of memory accesses. It enforces coherency between caches in the processors and system memory.
- **Write-back (WB)** — Writes and reads to and from system memory are cached. Reads come from cache lines on cache hits; read misses cause cache fills. Speculative reads are allowed. Write misses cause cache line fills (in processor families starting with the P6 family processors), and writes are performed entirely in the cache, when possible. Write combining is allowed. The write-back memory type reduces bus traffic by eliminating many unnecessary writes to system memory. Writes to a cache line are not immediately forwarded to system memory; instead, they are accumulated in the cache. The modified cache lines are written to system memory later, when a write-back operation is performed. Write-back operations are triggered when cache lines need to be deallocated, such as when new cache lines are being allocated in a cache that is already full. They also are triggered by the mechanisms used to maintain cache consistency. This type of cache-control provides the best performance, but it requires that all devices that access system memory on the system bus be able to snoop memory accesses to ensure system memory and cache coherency.
- **Write protected (WP) —** Reads come from cache lines when possible, and read misses cause cache fills. Writes are propagated to the system bus and cause corresponding cache lines on all processors on the bus to be invalidated. Speculative reads are allowed. This memory type is available in processor families starting from the P6 family processors by programming the MTRRs (see Table 13-6).

Table 13-3 shows which of these caching methods are available in the Pentium, P6 Family, Pentium 4, and Intel Xeon processors.

Table 13-3. Methods of Caching Available in Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, P6 Family, and Pentium Processors

Memory Type	Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4 and Intel Xeon Processors	P6 Family Processors	Pentium Processor
Strong Uncacheable (UC)	Yes	Yes	Yes
Uncacheable (UC-)	Yes	Yes*	No
Write Combining (WC)	Yes	Yes	No
Write Through (WT)	Yes	Yes	Yes
Write Back (WB)	Yes	Yes	Yes
Write Protected (WP)	Yes	Yes	No

NOTE:

* Introduced in the Pentium III processor; not available in the Pentium Pro or Pentium II processors

13.3.1 Buffering of Write Combining Memory Locations

Writes to the WC memory type are not cached in the typical sense of the word cached. They are retained in an internal write combining buffer (WC buffer) that is separate from the internal L1, L2, and L3 caches and the store buffer. The WC buffer is not snooped and thus does not provide data coherency. Buffering of writes to WC memory is done to allow software a small window of time to supply more modified data to the WC buffer while remaining as non-intrusive to software as possible. The buffering of writes to WC memory also causes data to be collapsed; that is, multiple writes to the same memory location will leave the last data written in the location and the other writes will be lost.

The size and structure of the WC buffer is not architecturally defined. For the Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4 and Intel Xeon processors; the WC buffer is made up of several 64-byte WC buffers. For the P6 family processors, the WC buffer is made up of several 32-byte WC buffers.

When software begins writing to WC memory, the processor begins filling the WC buffers one at a time. When one or more WC buffers has been filled, the processor has the option of evicting the buffers to system memory. The protocol for evicting the WC buffers is implementation dependent and should not be relied on by software for system memory coherency. When using the WC memory type, software **must** be sensitive to the fact that the writing of data to system memory is being delayed and **must** deliberately empty the WC buffers when system memory coherency is required.

Once the processor has started to evict data from the WC buffer into system memory, it will make a bus-transaction style decision based on how much of the buffer contains valid data. If the buffer is full (for example, all bytes are valid), the processor will execute a burst-write transaction on the bus. This results in all 32 bytes (P6 family processors) or 64 bytes (Pentium 4 and more recent processor) being transmitted on the data bus in a single burst transaction. If one or more of the WC buffer's bytes are invalid (for example, have not been written by software), the processor will transmit the data to memory using "partial write" transactions (one chunk at a time, where a "chunk" is 8 bytes).

This will result in a maximum of 4 partial write transactions (for P6 family processors) or 8 partial write transactions (for the Pentium 4 and more recent processors) for one WC buffer of data sent to memory.

The WC memory type is weakly ordered by definition. Once the eviction of a WC buffer has started, the data is subject to the weak ordering semantics of its definition. Ordering is not maintained between the successive allocation/deallocation of WC buffers (for example, writes to WC buffer 1 followed by writes to WC buffer 2 may appear as buffer 2 followed by buffer 1 on the system bus). When a WC buffer is evicted to memory as partial writes there is no guaranteed ordering between successive partial writes (for example, a partial write for chunk 2 may appear on the bus before the partial write for chunk 1 or vice versa).

The only elements of WC propagation to the system bus that are guaranteed are those provided by transaction atomicity. For example, with a P6 family processor, a completely full WC buffer will always be propagated as a single 32-bit burst transaction using any chunk order. In a WC buffer eviction where data will be evicted as partials, all data contained in the same chunk (0 mod 8 aligned) will be propagated simultaneously. Likewise, for more recent processors starting with those based on Intel NetBurst microarchitectures, a full WC buffer will always be propagated as a single burst transactions, using any chunk order within a transaction. For partial buffer propagations, all data contained in the same chunk will be propagated simultaneously.

13.3.2 Choosing a Memory Type

The simplest system memory model does not use memory-mapped I/O with read or write side effects, does not include a frame buffer, and uses the write-back memory type for all memory. An I/O agent can perform direct memory access (DMA) to write-back memory and the cache protocol maintains cache coherency.

A system can use strong uncacheable memory for other memory-mapped I/O, and should always use strong uncacheable memory for memory-mapped I/O with read side effects.

Dual-ported memory can be considered a write side effect, making relatively prompt writes desirable, because those writes cannot be observed at the other port until they reach the memory agent. A system can use strong uncacheable, uncacheable, write-through, or write-combining memory for frame buffers or dual-ported memory that contains pixel values displayed on a screen. Frame buffer memory is typically large (a few megabytes) and is usually written more than it is read by the processor. Using strong uncacheable memory for a frame buffer generates very large amounts of bus traffic, because operations on the entire buffer are implemented using partial writes rather than line writes. Using write-through memory for a frame buffer can displace almost all other useful cached

lines in the processor's L2 and L3 caches and L1 data cache. Therefore, systems should use write-combining memory for frame buffers whenever possible.

Software can use page-level cache control, to assign appropriate effective memory types when software will not access data structures in ways that benefit from write-back caching. For example, software may read a large data structure once and not access the structure again until the structure is rewritten by another agent. Such a large data structure should be marked as uncacheable, or reading it will evict cached lines that the processor will be referencing again.

A similar example would be a write-only data structure that is written to (to export the data to another agent), but never read by software. Such a structure can be marked as uncacheable, because software never reads the values that it writes (though as uncacheable memory, it will be written using partial writes, while as write-back memory, it will be written using line writes, which may not occur until the other agent reads the structure and triggers implicit write-backs).

On the Pentium III, Pentium 4, and more recent processors, new instructions are provided that give software greater control over the caching, prefetching, and the write-back characteristics of data. These instructions allow software to use weakly ordered or processor ordered memory types to improve processor performance, but when necessary to force strong ordering on memory reads and/or writes. They also allow software greater control over the caching of data. For a description of these instructions and their intended use, see Section 13.5.5, "Cache Management Instructions."

13.3.3 Code Fetches in Uncacheable Memory

Programs may execute code from uncacheable (UC) memory, but the implications are different from accessing data in UC memory. When doing code fetches, the processor never transitions from cacheable code to UC code speculatively. It also never speculatively fetches branch targets that result in UC code.

The processor may fetch the same UC cache line multiple times in order to decode an instruction once. It may decode consecutive UC instructions in a cache line without fetching between each instruction. It may also fetch additional cachelines from the same or a consecutive 4-KByte page in order to decode one non-speculative UC instruction (this can be true even when the instruction is contained fully in one line).

Because of the above and because cache line sizes may change in future processors, software should avoid placing memory-mapped I/O with read side effects in the same page or in a subsequent page used to execute UC code.

13.4 CACHE CONTROL PROTOCOL

The following section describes the cache control protocol currently defined for the Intel 64 and IA-32 architectures.

In the L1 data cache and in the L2/L3 unified caches, the MESI (modified, exclusive, shared, invalid) cache protocol maintains consistency with caches of other processors. The L1 data cache and the L2/L3 unified caches have two MESI status flags per cache line. Each line can be marked as being in one of the states defined in Table 13-4. In general, the operation of the MESI protocol is transparent to programs.

Table 13-4. MESI Cache Line States

Cache Line State	M (Modified)	E (Exclusive)	S (Shared)	I (Invalid)
This cache line is valid?	Yes	Yes	Yes	No
The memory copy is...	Out of date	Valid	Valid	—
Copies exist in caches of other processors?	No	No	Maybe	Maybe
A write to this line ...	Does not go to the system bus.	Does not go to the system bus.	Causes the processor to gain exclusive ownership of the line.	Goes directly to the system bus.

The L1 instruction cache in P6 family processors implements only the "SI" part of the MESI protocol, because the instruction cache is not writable. The instruction cache monitors changes in the data cache to maintain consistency between the caches when instructions are modified. See Section 13.6, "Self-Modifying Code," for more information on the implications of caching instructions.

13.5 CACHE CONTROL

The Intel 64 and IA-32 architectures provide a variety of mechanisms for controlling the caching of data and instructions and for controlling the ordering of reads and writes between the processor, the caches, and memory. These mechanisms can be divided into two groups:

- **Cache control registers and bits** — The Intel 64 and IA-32 architectures define several dedicated registers and various bits within control registers and page- and directory-table entries that control the caching system memory locations in the L1, L2, and L3 caches. These mechanisms control the caching of virtual memory pages and of regions of physical memory.
- **Cache control and memory ordering instructions** — The Intel 64 and IA-32 architectures provide several instructions that control the caching of data, the ordering of memory reads and writes, and the prefetching of data. These instructions allow software to control the caching of specific data structures, to control memory coherency for specific locations in memory, and to force strong memory ordering at specific locations in a program.

The following sections describe these two groups of cache control mechanisms.

13.5.1 Cache Control Registers and Bits

Figure 13-3 depicts cache-control mechanisms in IA-32 processors. Other than for the matter of memory address space, these work the same in Intel 64 processors.

The Intel 64 and IA-32 architectures provide the following cache-control registers and bits for use in enabling or restricting caching to various pages or regions in memory:

- **CD flag, bit 30 of control register CR0** — Controls caching of system memory locations (see Section 2.5, "Control Registers"). If the CD flag is clear, caching is enabled for the whole of system memory, but may be restricted for individual pages or regions of memory by other cache-control mechanisms. When the CD flag is set, caching is restricted in the processor's caches (cache hierarchy) for the P6 and more recent processor families and prevented for the Pentium processor (see note below). With the CD flag set, however, the caches will still respond to snoop traffic. Caches should be explicitly flushed to ensure memory coherency. For highest processor performance, both the CD and the NW flags in control register CR0 should be cleared. Table 13-5 shows the interaction of the CD and NW flags.

  The effect of setting the CD flag is somewhat different for processor families starting with P6 family than the Pentium processor (see Table 13-5). To ensure memory coherency after the CD flag is set, the caches should be explicitly flushed (see Section 13.5.3, "Preventing Caching"). Setting the CD flag for the P6 and more recent processor families modifies cache line fill and update behavior. Also, setting the CD flag on these processors do not force strict ordering of memory accesses unless the MTRRs are disabled and/or all memory is referenced as uncached (see Section 10.2.5, "Strengthening or Weakening the Memory-Ordering Model").

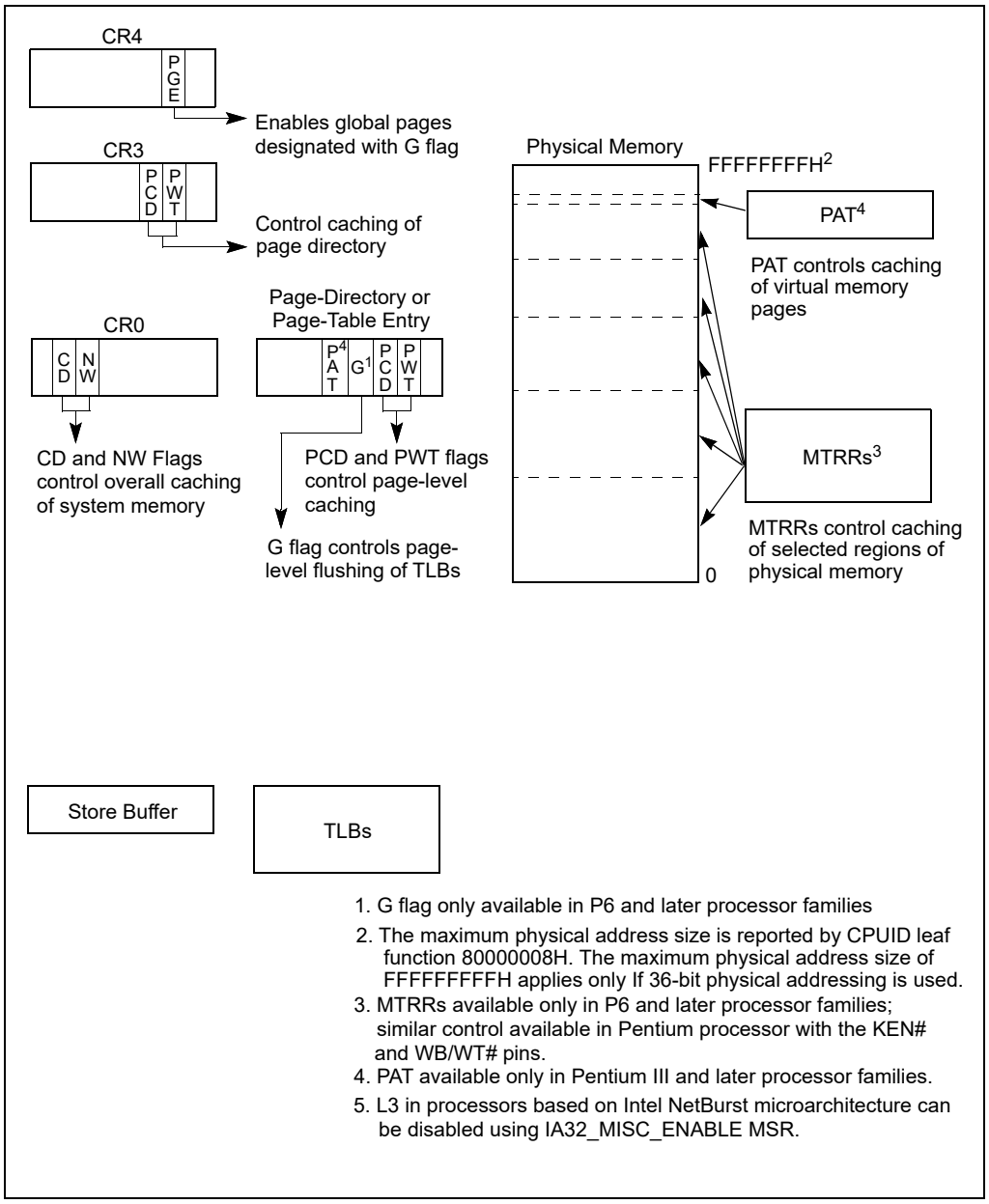


Figure 13-3. Cache-Control Registers and Bits Available in Intel 64 and IA-32 Processors

Table 13-5. Cache Operating Modes

CD	NW	Caching and Read/Write Policy	L1	L2/L3[1]
0	0	Normal Cache Mode. Highest performance cache operation.		
		▪ Read hits access the cache; read misses may cause replacement.	Yes	Yes
		▪ Write hits update the cache.	Yes	Yes
		▪ Only writes to shared lines and write misses update system memory.	Yes	Yes
		▪ Write misses cause cache line fills.	Yes	Yes
		▪ Write hits can change shared lines to modified under control of the MTRRs and with associated read invalidation cycle.	Yes	
		▪ (Pentium processor only.) Write misses do not cause cache line fills.	Yes	
		▪ (Pentium processor only.) Write hits can change shared lines to exclusive under control of WB/WT#.	Yes	
		▪ Invalidation is allowed.		
		▪ External snoop traffic is supported.	Yes	Yes
			Yes	Yes
0	1	Invalid setting.		
		Generates a general-protection exception (#GP) with an error code of 0.	NA	NA
1	0	No-fill Cache Mode. Memory coherency is maintained.[3]		
		▪ (Pentium 4 and later processor families.) State of processor after a power up or reset.	Yes	Yes
		▪ Read hits access the cache; read misses do not cause replacement (see Pentium 4 and Intel Xeon processors reference below).	Yes	Yes
		▪ Write hits update the cache.		
		▪ Only writes to shared lines and write misses update system memory.	Yes	Yes
			Yes	Yes
		▪ Write misses access memory.	Yes	Yes
		▪ Write hits can change shared lines to exclusive under control of the MTRRs and with associated read invalidation cycle.	Yes	Yes
		▪ (Pentium processor only.) Write hits can change shared lines to exclusive under control of the WB/WT#.	Yes	
		▪ (P6 and later processor families only.) Strict memory ordering is not enforced unless the MTRRs are disabled and/or all memory is referenced as uncached (see Section 7.2.4., "Strengthening or Weakening the Memory Ordering Model").	Yes	Yes
		▪ Invalidation is allowed.		
		▪ External snoop traffic is supported.	Yes	Yes
			Yes	Yes
1	1	Memory coherency is not maintained.[2, 3]		
		▪ (P6 family and Pentium processors.) State of the processor after a power up or reset.	Yes	Yes
		▪ Read hits access the cache; read misses do not cause replacement.		
		▪ Write hits update the cache and change exclusive lines to modified.	Yes	Yes
			Yes	Yes
		▪ Shared lines remain shared after write hit.	Yes	Yes
		▪ Write misses access memory.	Yes	Yes
		▪ Invalidation is inhibited when snooping; but is allowed with INVD and WBINVD instructions.	Yes	Yes
		▪ External snoop traffic is supported.		
			No	Yes

NOTES:

1. The L2/L3 column in this table is definitive for the Pentium 4, Intel Xeon, and P6 family processors. It is intended to represent what could be implemented in a system based on a Pentium processor with an external, platform specific, write-back L2 cache.
2. The Pentium 4 and more recent processor families do not support this mode; setting the CD and NW bits to 1 selects the no-fill cache mode.
3. Not supported In Intel Atom processors. If CD = 1 in an Intel Atom processor, caching is disabled.

- **NW flag, bit 29 of control register CR0** — Controls the write policy for system memory locations (see Section 2.5, "Control Registers"). If the NW and CD flags are clear, write-back is enabled for the whole of system memory, but may be restricted for individual pages or regions of memory by other cache-control mechanisms. Table 13-5 shows how the other combinations of CD and NW flags affects caching.

NOTES

For the Pentium 4 and Intel Xeon processors, the NW flag is a don't care flag; that is, when the CD flag is set, the processor uses the no-fill cache mode, regardless of the setting of the NW flag.

For Intel Atom processors, the NW flag is a don't care flag; that is, when the CD flag is set, the processor disables caching, regardless of the setting of the NW flag.

For the Pentium processor, when the L1 cache is disabled (the CD and NW flags in control register CR0 are set), external snoops are accepted in DP (dual-processor) systems and inhibited in uniprocessor systems.

When snoops are inhibited, address parity is not checked and APCHK# is not asserted for a corrupt address; however, when snoops are accepted, address parity is checked and APCHK# is asserted for corrupt addresses.

- **PCD and PWT flags in paging-structure entries** — Control the memory type used to access paging structures and pages (see Section 5.9, "Paging and Memory Typing").
- **PCD and PWT flags in control register CR3** — Control the memory type used to access the first paging structure of the current paging-structure hierarchy (see Section 5.9, "Paging and Memory Typing").
- **G (global) flag in the page-directory and page-table entries (introduced to the IA-32 architecture in the P6 family processors)** — Controls the flushing of TLB entries for individual pages. See Section 5.10, "Caching Translation Information," for more information about this flag.
- **PGE (page global enable) flag in control register CR4** — Enables the establishment of global pages with the G flag. See Section 5.10, "Caching Translation Information," for more information about this flag.
- **Memory type range registers (MTRRs) (introduced in P6 family processors)** — Control the type of caching used in specific regions of physical memory. Any of the caching types described in Section 13.3, "Methods of Caching Available," can be selected. See Section 13.11, "Memory Type Range Registers (MTRRs)," for a detailed description of the MTRRs.
- **Page Attribute Table (PAT) MSR (introduced in the Pentium III processor)** — Extends the memory typing capabilities of the processor to permit memory types to be assigned on a page-by-page basis (see Section 13.12, "Page Attribute Table (PAT)").
- **Third-Level Cache Disable flag, bit 6 of the IA32_MISC_ENABLE MSR (Available only in processors based on Intel NetBurst microarchitecture)** — Allows the L3 cache to be disabled and enabled, independently of the L1 and L2 caches.
- **KEN# and WB/WT# pins (Pentium processor)** — Allow external hardware to control the caching method used for specific areas of memory. They perform similar (but not identical) functions to the MTRRs in the P6 family processors.
- **PCD and PWT pins (Pentium processor)** — These pins (which are associated with the PCD and PWT flags in control register CR3 and in the page-directory and page-table entries) permit caching in an external L2 cache to be controlled on a page-by-page basis, consistent with the control exercised on the L1 cache of these processors. The P6 and more recent processor families do not provide these pins because the L2 cache in internal to the chip package.

13.5.2 Precedence of Cache Controls

The cache control flags and MTRRs operate hierarchically for restricting caching. That is, if the CD flag is set, caching is prevented globally (see Table 13-5). If the CD flag is clear, the page-level cache control flags and/or the MTRRs can be used to restrict caching. If there is an overlap of page-level and MTRR caching controls, the mechanism that prevents caching has precedence. For example, if an MTRR makes a region of system memory uncacheable, a page-level caching control cannot be used to enable caching for a page in that region. The converse is also

true; that is, if a page-level caching control designates a page as uncacheable, an MTRR cannot be used to make the page cacheable.

In cases where there is a overlap in the assignment of the write-back and write-through caching policies to a page and a region of memory, the write-through policy takes precedence. The write-combining policy (which can only be assigned through an MTRR or the PAT) takes precedence over either write-through or write-back.

The selection of memory types at the page level varies depending on whether PAT is being used to select memory types for pages, as described in the following sections.

On processors based on Intel NetBurst microarchitecture, the third-level cache can be disabled by bit 6 of the IA32_MISC_ENABLE MSR. Using IA32_MISC_ENABLE[bit 6] takes precedence over the CD flag, MTRRs, and PAT for the L3 cache in those processors. That is, when the third-level cache disable flag is set (cache disabled), the other cache controls have no affect on the L3 cache; when the flag is clear (enabled), the cache controls have the same affect on the L3 cache as they have on the L1 and L2 caches.

IA32_MISC_ENABLE[bit 6] is not supported in Intel Core i7 processors, nor processors based on Intel Core, and Intel Atom microarchitectures.

13.5.2.1 Selecting Memory Types for Pentium Pro and Pentium II Processors

The Pentium Pro and Pentium II processors do not support the PAT. Here, the effective memory type for a page is selected with the MTRRs and the PCD and PWT bits in the page-table or page-directory entry for the page. Table 13-6 describes the mapping of MTRR memory types and page-level caching attributes to effective memory types, when normal caching is in effect (the CD and NW flags in control register CR0 are clear). Combinations that appear in gray are implementation-defined for the Pentium Pro and Pentium II processors. System designers are encouraged to avoid these implementation-defined combinations.

Table 13-6. Effective Page-Level Memory Type for Pentium Pro and Pentium II Processors

MTRR Memory Type[1]	PCD Value	PWT Value	Effective Memory Type
UC	X	X	UC
WC	0	0	WC
	0	1	WC
	1	0	WC
	1	1	UC
WT	0	X	WT
	1	X	UC
WP	0	0	WP
	0	1	WP
	1	0	WC
	1	1	UC
WB	0	0	WB
	0	1	WT
	1	X	UC

NOTE:

1. These effective memory types also apply to the Pentium 4, Intel Xeon, and Pentium III processors when the PAT bit is not used (set to 0) in page-table and page-directory entries.

When normal caching is in effect, the effective memory type shown in Table 13-6 is determined using the following rules:

1. If the PCD and PWT attributes for the page are both 0, then the effective memory type is identical to the MTRR-defined memory type.

2. If the PCD flag is set, then the effective memory type is UC.
3. If the PCD flag is clear and the PWT flag is set, the effective memory type is WT for the WB memory type and the MTRR-defined memory type for all other memory types.
4. Setting the PCD and PWT flags to opposite values is considered model-specific for the WP and WC memory types and architecturally-defined for the WB, WT, and UC memory types.

13.5.2.2 Selecting Memory Types for Pentium III and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Intel Core Solo, Pentium M, Pentium 4, Intel Xeon, and Pentium III processors use the PAT to select effective page-level memory types. Here, a memory type for a page is selected by the MTRRs and the value in a PAT entry that is selected with the PAT, PCD, and PWT bits in a page-table or page-directory entry (see Section 13.12.3, "Selecting a Memory Type from the PAT"). Table 13-7 describes the mapping of MTRR memory types and PAT entry types to effective memory types, when normal caching is in effect (the CD and NW flags in control register CR0 are clear).

Table 13-7. Effective Page-Level Memory Types for Pentium III and More Recent Processor Families

MTRR Memory Type	PAT Entry Value	Effective Memory Type
UC	UC	UC[1]
	UC-	UC[1]
	WC	WC
	WT	UC[1]
	WB	UC[1]
	WP	UC[1]
WC	UC	UC[2]
	UC-	WC
	WC	WC
	WT	UC[2,3]
	WB	WC
	WP	UC[2,3]
WT	UC	UC[2]
	UC-	UC[2]
	WC	WC
	WT	WT
	WB	WT
	WP	WP[3]
WB	UC	UC[2]
	UC-	UC[2]
	WC	WC
	WT	WT
	WB	WB
	WP	WP

Table 13-7. Effective Page-Level Memory Types for Pentium III and More Recent Processor Families (Contd.)

MTRR Memory Type	PAT Entry Value	Effective Memory Type
WP	UC	UC[2]
	UC-	WC[3]
	WC	WC
	WT	WT[3]
	WB	WP
	WP	WP

NOTES:

1. The UC attribute comes from the MTRRs and the processors are not required to snoop their caches since the data could never have been cached. This attribute is preferred for performance reasons.
2. The UC attribute came from the page-table or page-directory entry and processors are required to check their caches because the data may be cached due to page aliasing, which is not recommended.
3. These combinations were specified as "undefined" in previous editions of the Intel® 64 and IA-32 Architectures Software Developer's Manual. However, all processors that support both the PAT and the MTRRs determine the effective page-level memory types for these combinations as given.

13.5.2.3 Writing Values Across Pages with Different Memory Types

If two adjoining pages in memory have different memory types, and a word or longer operand is written to a memory location that crosses the page boundary between those two pages, the operand might be written to memory twice. This action does not present a problem for writes to actual memory; however, if a device is mapped to the memory space assigned to the pages, the device might malfunction.

13.5.3 Preventing Caching

To disable the L1, L2, and L3 caches after they have been enabled and have received cache fills, perform the following steps:

1. Enter the no-fill cache mode. (Set the CD flag in control register CR0 to 1 and the NW flag to 0.
2. Flush all caches using the WBINVD instruction.
3. Disable the MTRRs and set the default memory type to uncached or set all MTRRs for the uncached memory type (see the discussion of the discussion of the TYPE field and the E flag in Section 13.11.2.1, "IA32_MTRR_DEF_TYPE MSR").

The caches must be flushed (step 2) after the CD flag is set to ensure system memory coherency. If the caches are not flushed, cache hits on reads will still occur and data will be read from valid cache lines.

The intent of the three separate steps listed above address three distinct requirements: (i) discontinue new data replacing existing data in the cache (ii) ensure data already in the cache are evicted to memory, (iii) ensure subsequent memory references observe UC memory type semantics. Different processor implementation of caching control hardware may allow some variation of software implementation of these three requirements. See note below.

NOTES

Setting the CD flag in control register CR0 modifies the processor's caching behavior as indicated in Table 13-5, but setting the CD flag alone may not be sufficient across all processor families to force the effective memory type for all physical memory to be UC nor does it force strict memory ordering, due to hardware implementation variations across different processor families. To force the UC memory type and strict memory ordering on all of physical memory, it is sufficient to either program the MTRRs for all physical memory to be UC memory type or disable all MTRRs.

For the Pentium 4 and Intel Xeon processors, after the sequence of steps given above has been executed, the cache lines containing the code between the end of the WBINVD instruction and before the MTRRS have actually been disabled may be retained in the cache hierarchy. Here, to

remove code from the cache completely, a second WBINVD instruction must be executed after the MTRRs have been disabled.

For Intel Atom processors, setting the CD flag forces all physical memory to observe UC semantics (without requiring memory type of physical memory to be set explicitly). Consequently, software does not need to issue a second WBINVD as some other processor generations might require.

13.5.4 Disabling and Enabling the L3 Cache

On processors based on Intel NetBurst microarchitecture, the third-level cache can be disabled by bit 6 of the IA32_MISC_ENABLE MSR. The third-level cache disable flag (bit 6 of the IA32_MISC_ENABLE MSR) allows the L3 cache to be disabled and enabled, independently of the L1 and L2 caches. Prior to using this control to disable or enable the L3 cache, software should disable and flush all the processor caches, as described earlier in Section 13.5.3, "Preventing Caching," to prevent of loss of information stored in the L3 cache. After the L3 cache has been disabled or enabled, caching for the whole processor can be restored.

Newer Intel 64 processor with L3 do not support IA32_MISC_ENABLE[bit 6], the procedure described in Section 13.5.3, "Preventing Caching," apply to the entire cache hierarchy.

13.5.5 Cache Management Instructions

The Intel 64 and IA-32 architectures provide several instructions for managing the L1, L2, and L3 caches. The INVD and WBINVD instructions are privileged instructions and operate on the L1, L2, and L3 caches as a whole. The PREFETCHh, CLFLUSH, and CLFLUSHOPT instructions and the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD) offer more granular control over caching, and are available to all privileged levels.

The INVD and WBINVD instructions are used to invalidate the contents of the L1, L2, and L3 caches. The INVD instruction invalidates all internal cache entries, then generates a special-function bus cycle that indicates that external caches also should be invalidated. The INVD instruction should be used with care. It does not force a write-back of modified cache lines; therefore, data stored in the caches and not written back to system memory will be lost. Unless there is a specific requirement or benefit to invalidating the caches without writing back the modified lines (such as, during testing or fault recovery where cache coherency with main memory is not a concern), software should use the WBINVD instruction.

The WBINVD instruction first writes back any modified lines in all the internal caches, then invalidates the contents of the L1, L2, and L3 caches. It ensures that cache coherency with main memory is maintained regardless of the write policy in effect (that is, write-through or write-back). Following this operation, the WBINVD instruction generates one (P6 family processors) or two (Pentium and Intel486 processors) special-function bus cycles to indicate to external cache controllers that write-back of modified data followed by invalidation of external caches should occur. The amount of time or cycles for WBINVD to complete will vary due to the size of different cache hierarchies and other factors. As a consequence, the use of the WBINVD instruction can have an impact on interrupt/event response time.

The PREFETCH*h* instructions allow a program to suggest to the processor that a cache line from a specified location in system memory be prefetched into the cache hierarchy (see Section 13.8, "Explicit Caching").

The CLFLUSH and CLFLUSHOPT instructions allow selected cache lines to be flushed from memory. These instructions give a program the ability to explicitly free up cache space, when it is known that cached section of system memory will not be accessed in the near future.

The non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD) allow data to be moved from the processor's registers directly into system memory without being also written into the L1, L2, and/or L3 caches. These instructions can be used to prevent cache pollution when operating on data that is going to be modified only once before being stored back into system memory. These instructions operate on data in the general-purpose, MMX, and XMM registers.

13.5.6 L1 Data Cache Context Mode

L1 data cache context mode is a feature of processors based on the Intel NetBurst microarchitecture that support Intel Hyper-Threading Technology. When CPUID.1:ECX[bit 10] = 1, the processor supports setting L1 data cache context mode using the L1 data cache context mode flag (IA32_MISC_ENABLE[bit 24]). Selectable modes are adaptive mode (default) and shared mode.

The BIOS is responsible for configuring the L1 data cache context mode.

13.5.6.1 Adaptive Mode

Adaptive mode facilitates L1 data cache sharing between logical processors. When running in adaptive mode, the L1 data cache is shared across logical processors in the same core if:

- CR3 control registers for logical processors sharing the cache are identical.
- The same paging mode is used by logical processors sharing the cache.

In this situation, the entire L1 data cache is available to each logical processor (instead of being competitively shared).

If CR3 values are different for the logical processors sharing an L1 data cache or the logical processors use different paging modes, processors compete for cache resources. This reduces the effective size of the cache for each logical processor. Aliasing of the cache is not allowed (which prevents data thrashing).

13.5.6.2 Shared Mode

In shared mode, the L1 data cache is competitively shared between logical processors. This is true even if the logical processors use identical CR3 registers and paging modes.

In shared mode, linear addresses in the L1 data cache can be aliased, meaning that one linear address in the cache can point to different physical locations. The mechanism for resolving aliasing can lead to thrashing. For this reason, IA32_MISC_ENABLE[bit 24] = 0 is the preferred configuration for processors based on the Intel NetBurst microarchitecture that support Intel Hyper-Threading Technology.

13.6 SELF-MODIFYING CODE

A write to a memory location in a code segment that is currently cached in the processor causes the associated cache line (or lines) to be invalidated. This check is based on the physical address of the instruction. In addition, the P6 family and Pentium processors check whether a write to a code segment may modify an instruction that has been prefetched for execution. If the write affects a prefetched instruction, the prefetch queue is invalidated. This latter check is based on the linear address of the instruction. For the Pentium 4 and Intel Xeon processors, a write or a snoop of an instruction in a code segment, where the target instruction is already decoded and resident in the trace cache, invalidates the entire trace cache. The latter behavior means that programs that self-modify code can cause severe degradation of performance when run on the Pentium 4 and Intel Xeon processors.

In practice, the check on linear addresses should not create compatibility problems among IA-32 processors. Applications that include self-modifying code use the same linear address for modifying and fetching the instruction. Systems software, such as a debugger, that might possibly modify an instruction using a different linear address than that used to fetch the instruction, will execute a serializing operation, such as a CPUID instruction, before the modified instruction is executed, which will automatically resynchronize the instruction cache and prefetch queue. (See Section 10.1.3, “Handling Self- and Cross-Modifying Code,” for more information about the use of self-modifying code.)

For Intel486 processors, a write to an instruction in the cache will modify it in both the cache and memory, but if the instruction was prefetched before the write, the old version of the instruction could be the one executed. To prevent the old instruction from being executed, flush the instruction prefetch unit by coding a jump instruction immediately after any write that modifies an instruction.

13.7 IMPLICIT CACHING (PENTIUM 4, INTEL® XEON®, AND P6 FAMILY PROCESSORS)

Implicit caching occurs when a memory element is made potentially cacheable, although the element may never have been accessed in the normal von Neumann sequence. Implicit caching occurs on the P6 and more recent processor families due to aggressive prefetching, branch prediction, and TLB miss handling. Implicit caching is an extension of the behavior of existing Intel386, Intel486, and Pentium processor systems, since software running on these processor families also has not been able to deterministically predict the behavior of instruction prefetch.

To avoid problems related to implicit caching, the operating system must explicitly invalidate the cache when changes are made to cacheable data that the cache coherency mechanism does not automatically handle. This includes writes to dual-ported or physically aliased memory boards that are not detected by the snooping mechanisms of the processor, and changes to page- table entries in memory.

The code in Example 13-1 shows the effect of implicit caching on page-table entries. The linear address F000H points to physical location B000H (the page-table entry for F000H contains the value B000H), and the page-table entry for linear address F000 is PTE_F000.

Example 13-1. Effect of Implicit Caching on Page-Table Entries

```
mov EAX, CR3; Invalidate the TLB
mov CR3, EAX; by copying CR3 to itself
mov PTE_F000, A000H; Change F000H to point to A000H
mov EBX, [F000H];
```

Because of speculative execution in the P6 and more recent processor families, the last MOV instruction performed would place the value at physical location B000H into EBX, rather than the value at the new physical address A000H. This situation is remedied by placing a TLB invalidation between the load and the store.

13.8 EXPLICIT CACHING

The Pentium III processor introduced four new instructions, the PREFETCH*h* instructions, that provide software with explicit control over the caching of data. These instructions provide "hints" to the processor that the data requested by a PREFETCH*h* instruction should be read into cache hierarchy now or as soon as possible, in anticipation of its use. The instructions provide different variations of the hint that allow selection of the cache level into which data will be read.

The PREFETCH*h* instructions can help reduce the long latency typically associated with reading data from memory and thus help prevent processor "stalls." However, these instructions should be used judiciously. Overuse can lead to resource conflicts and hence reduce the performance of an application. Also, these instructions should only be used to prefetch data from memory; they should not be used to prefetch instructions. For more detailed information on the proper use of the prefetch instruction, refer to Chapter 7, "Optimizing Cache Usage," in the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

13.9 INVALIDATING THE TRANSLATION LOOKASIDE BUFFERS (TLBS)

The processor updates its address translation caches (TLBs) transparently to software. Several mechanisms are available, however, that allow software and hardware to invalidate the TLBs either explicitly or as a side effect of another operation. Most details are given in Section 5.10.4, "Invalidation of TLBs and Paging-Structure Caches." In addition, the following operations invalidate all TLB entries, irrespective of the setting of the G flag:

- Asserting or de-asserting the FLUSH# pin.
- (Pentium 4, Intel Xeon, and later processors only.) Writing to an MTRR (with a WRMSR instruction).
- Writing to control register CR0 to modify the PG or PE flag.

- (Pentium 4, Intel Xeon, and later processors only.) Writing to control register CR4 to modify the PSE, PGE, or PAE flag.
- Writing to control register CR4 to change the PCIDE flag from 1 to 0.

See Section 5.10, "Caching Translation Information," for additional information about the TLBs.

13.10 STORE BUFFER

Intel 64 and IA-32 processors temporarily store each write (store) to memory in a store buffer. The store buffer improves processor performance by allowing the processor to continue executing instructions without having to wait until a write to memory and/or to a cache is complete. It also allows writes to be delayed for more efficient use of memory-access bus cycles.

In general, the existence of the store buffer is transparent to software, even in systems that use multiple processors. The processor ensures that write operations are always carried out in program order. It also ensures that the contents of the store buffer are always drained to memory in the following situations:

- When an exception or interrupt is generated.
- (P6 and more recent processor families only) When a serializing instruction is executed.
- When an I/O instruction is executed.
- When a LOCK operation is performed.
- (P6 and more recent processor families only) When a BINIT operation is performed.
- (Pentium III, and more recent processor families only) When using an SFENCE instruction to order stores.
- (Pentium 4 and more recent processor families only) When using an MFENCE instruction to order stores.

The discussion of write ordering in Section 10.2, "Memory Ordering," gives a detailed description of the operation of the store buffer.

13.11 MEMORY TYPE RANGE REGISTERS (MTRRS)

The following section pertains only to the P6 and more recent processor families.

The memory type range registers (MTRRs) provide a mechanism for associating the memory types (see Section 13.3, "Methods of Caching Available") with physical-address ranges in system memory. They allow the processor to optimize operations for different types of memory such as RAM, ROM, frame-buffer memory, and memory-mapped I/O devices. They also simplify system hardware design by eliminating the memory control pins used for this function on earlier IA-32 processors and the external logic needed to drive them.

The MTRR mechanism allows multiple ranges to be defined in physical memory, and it defines a set of model-specific registers (MSRs) for specifying the type of memory that is contained in each range. Table 13-8 shows the memory types that can be specified and their properties; Figure 13-4 shows the mapping of physical memory with MTRRs. See Section 13.3, "Methods of Caching Available," for a more detailed description of each memory type.

Following a hardware reset, the P6 and more recent processor families disable all the fixed and variable MTRRs, which in effect makes all of physical memory uncacheable. Initialization software should then set the MTRRs to a specific, system-defined memory map. Typically, the BIOS (basic input/output system) software configures the MTRRs. The operating system or executive is then free to modify the memory map using the normal page-level cacheability attributes.

In a multiprocessor system using a processor in the P6 family or a more recent family, each processor MUST use the identical MTRR memory map so that software will have a consistent view of memory.

NOTE

In multiple processor systems, the operating system must maintain MTRR consistency between all the processors in the system (that is, all processors must use the same MTRR values). The P6 and more recent processor families provide no hardware support for maintaining this consistency.

Table 13-8. Memory Types That Can Be Encoded in MTRRs

Memory Type and Mnemonic	Encoding in MTRR
Uncacheable (UC)	00H
Write Combining (WC)	01H
Reserved*	02H
Reserved*	03H
Write-through (WT)	04H
Write-protected (WP)	05H
Writeback (WB)	06H
Reserved*	7H through FFH

NOTE:

* Use of these encodings results in a general-protection exception (#GP).

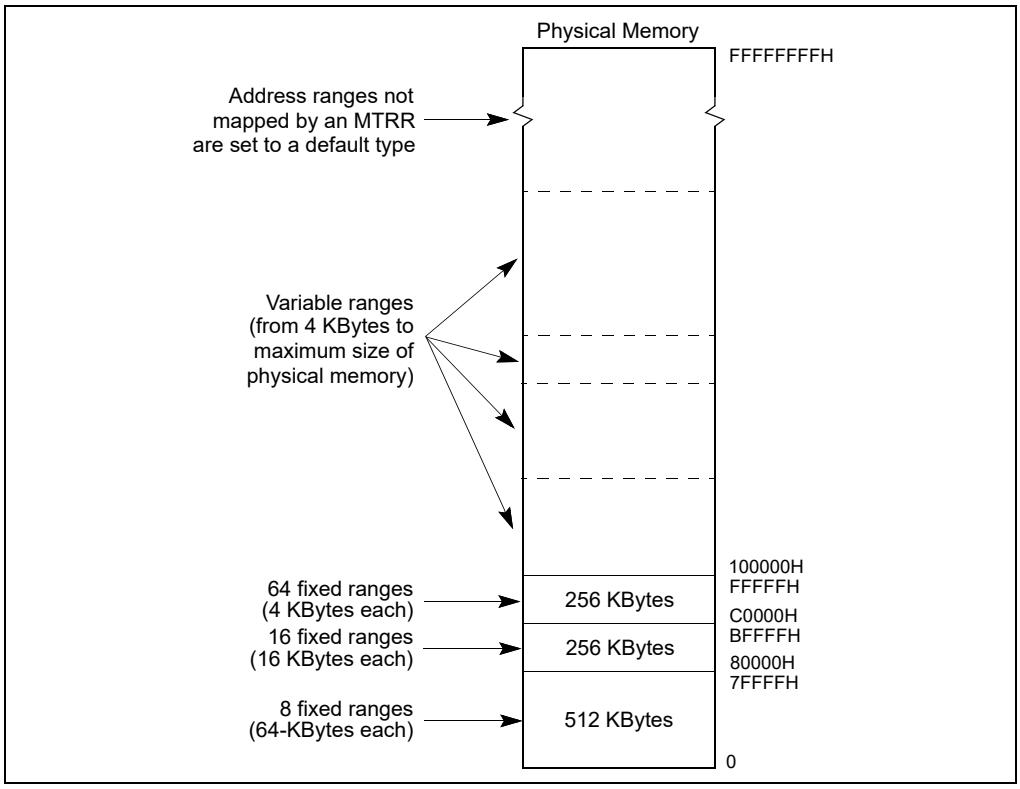


Figure 13-4. Mapping Physical Memory With MTRRs

13.11.1 MTRR Feature Identification

The availability of the MTRR feature is model-specific. Software can determine if MTRRs are supported on a processor by executing the CPUID instruction and reading the state of the MTRR flag (bit 12) in the feature information register (EDX).

If the MTRR flag is set (indicating that the processor implements MTRRs), additional information about MTRRs can be obtained from the 64-bit IA32_MTRRCAP MSR (named MTRRcap MSR for the P6 family processors). The IA32_MTRRCAP MSR is a read-only MSR that can be read with the RDMSR instruction. Figure 13-5 shows the contents of the IA32_MTRRCAP MSR. The functions of the flags and field in this register are as follows:

- **VCNT (variable range registers count) field, bits 0 through 7** — Indicates the number of variable ranges implemented on the processor.
- **FIX (fixed range registers supported) flag, bit 8** — Fixed range MTRRs (IA32_MTRR_FIX64K_00000 through IA32_MTRR_FIX4K_0F8000) are supported when set; no fixed range registers are supported when clear.
- **WC (write combining) flag, bit 10** — The write-combining (WC) memory type is supported when set; the WC type is not supported when clear.
- **SMRR (System-Management Range Register) flag, bit 11** — The system-management range register (SMRR) interface is supported when bit 11 is set; the SMRR interface is not supported when clear.

Bit 9 and bits 12 through 63 in the IA32_MTRRCAP MSR are reserved. If software attempts to write to the IA32_MTRRCAP MSR, a general-protection exception (#GP) is generated.

Software must read IA32_MTRRCAP VCNT field to determine the number of variable MTRRs and query other feature bits in IA32_MTRRCAP to determine additional capabilities that are supported in a processor. For example, some processors may report a value of '8' in the VCNT field, other processors may report VCNT with different values.

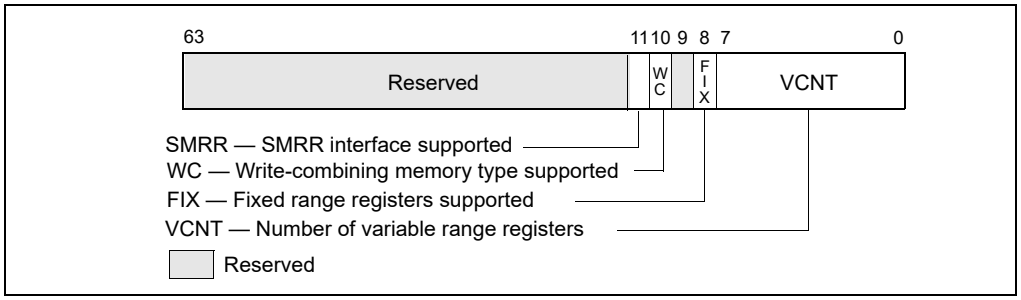


Figure 13-5. IA32_MTRRCAP Register

13.11.2 Setting Memory Ranges with MTRRs

The memory ranges and the types of memory specified in each range are set by three groups of registers: the IA32_MTRR_DEF_TYPE MSR, the fixed-range MTRRs, and the variable range MTRRs. These registers can be read and written to using the RDMSR and WRMSR instructions, respectively. The IA32_MTRRCAP MSR indicates the availability of these registers on the processor (see Section 13.11.1, "MTRR Feature Identification").

13.11.2.1 IA32_MTRR_DEF_TYPE MSR

The IA32_MTRR_DEF_TYPE MSR (named MTRRdefType MSR for the P6 family processors) sets the default properties of the regions of physical memory that are not encompassed by MTRRs. The functions of the flags and field in this register are as follows:

- **Type field, bits 0 through 7** — Indicates the default memory type used for those physical memory address ranges that do not have a memory type specified for them by an MTRR (see Table 13-8 for the encoding of this field). The legal values for this field are 0, 1, 4, 5, and 6. All other values result in a general-protection exception (#GP) being generated.

  Intel recommends the use of the UC (uncached) memory type for all physical memory addresses where memory does not exist. To assign the UC type to nonexistent memory locations, it can either be specified as the default type in the Type field or be explicitly assigned with the fixed and variable MTRRs.

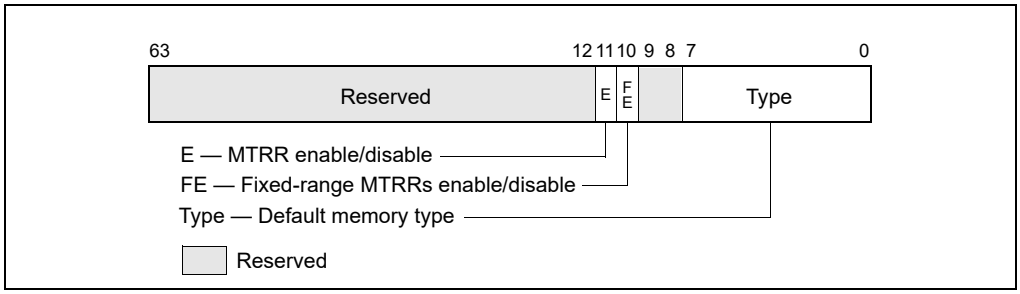


Figure 13-6. IA32_MTRR_DEF_TYPE MSR

- **FE (fixed MTRRs enabled) flag, bit 10** — Fixed-range MTRRs are enabled when set; fixed-range MTRRs are disabled when clear. When the fixed-range MTRRs are enabled, they take priority over the variable-range MTRRs when overlaps in ranges occur. If the fixed-range MTRRs are disabled, the variable-range MTRRs can still be used and can map the range ordinarily covered by the fixed-range MTRRs.
- **E (MTRRs enabled) flag, bit 11** — MTRRs are enabled when set; all MTRRs are disabled when clear, and the UC memory type is applied to all of physical memory. When this flag is set, the FE flag can disable the fixed-range MTRRs; when the flag is clear, the FE flag has no affect. When the E flag is set, the type specified in the default memory type field is used for areas of memory not already mapped by either a fixed or variable MTRR.

Bits 8 and 9, and bits 12 through 63, in the IA32_MTRR_DEF_TYPE MSR are reserved; the processor generates a general-protection exception (#GP) if software attempts to write nonzero values to them.

13.11.2.2 Fixed Range MTRRs

The fixed memory ranges are mapped with 11 fixed-range registers of 64 bits each. Each of these registers is divided into 8-bit fields that are used to specify the memory type for each of the sub-ranges the register controls:

- **Register IA32_MTRR_FIX64K_00000** — Maps the 512-KByte address range from 0H to 7FFFFH. This range is divided into eight 64-KByte sub-ranges.
- **Registers IA32_MTRR_FIX16K_80000 and IA32_MTRR_FIX16K_A0000** — Maps the two 128-KByte address ranges from 80000H to BFFFFH. This range is divided into sixteen 16-KByte sub-ranges, 8 ranges per register.
- **Registers IA32_MTRR_FIX4K_C0000 through IA32_MTRR_FIX4K_F8000** — Maps eight 32-KByte address ranges from C0000H to FFFFFH. This range is divided into sixty-four 4-KByte sub-ranges, 8 ranges per register.

Table 13-9 shows the relationship between the fixed physical-address ranges and the corresponding fields of the fixed-range MTRRs; Table 13-8 shows memory type encoding for MTRRs.

For the P6 family processors, the prefix for the fixed range MTRRs is MTRRfix.

13.11.2.3 Variable Range MTRRs

The Pentium 4, Intel Xeon, and P6 family processors permit software to specify the memory type for m variable-size address ranges, using a pair of MTRRs for each range. The number m of ranges supported is given in bits 7:0 of the IA32_MTRRCAP MSR (see Figure 13-5 in Section 13.11.1).

The first entry in each pair (IA32_MTRR_PHYSBASEn) defines the base address and memory type for the range; the second entry (IA32_MTRR_PHYSMASKn) contains a mask used to determine the address range. The “n” suffix is in the range 0 through m–1 and identifies a specific register pair.

For P6 family processors, the prefixes for these variable range MTRRs are MTRRphysBase and MTRRphysMask.

Table 13-9. Address Mapping for Fixed-Range MTRRs

Address Range (hexadecimal)								MTRR
63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0	
70000-7FFFF	60000-6FFFF	50000-5FFFF	40000-4FFFF	30000-3FFFF	20000-2FFFF	10000-1FFFF	00000-0FFFF	IA32_MTRR_FIX64K_00000
9C000 9FFFF	98000-9BFFF	94000-97FFF	90000-93FFF	8C000-8FFFF	88000-8BFFF	84000-87FFF	80000-83FFF	IA32_MTRR_FIX16K_80000
BC000 BFFFF	B8000-BBFFF	B4000-B7FFF	B0000-B3FFF	AC000-AFFFF	A8000-ABFFF	A4000-A7FFF	A0000-A3FFF	IA32_MTRR_FIX16K_A0000
C7000 C7FFF	C6000-C6FFF	C5000-C5FFF	C4000-C4FFF	C3000-C3FFF	C2000-C2FFF	C1000-C1FFF	C0000-C0FFF	IA32_MTRR_FIX4K_C0000
CF000 CFFFF	CE000-CEFFF	CD000-CDFFF	CC000-CCFFF	CB000-CBFFF	CA000-CAFFF	C9000-C9FFF	C8000-C8FFF	IA32_MTRR_FIX4K_C8000
D7000 D7FFF	D6000-D6FFF	D5000-D5FFF	D4000-D4FFF	D3000-D3FFF	D2000-D2FFF	D1000-D1FFF	D0000-D0FFF	IA32_MTRR_FIX4K_D0000
DF000 DFFFF	DE000-DEFFF	DD000-DDFFF	DC000-DCFFF	DB000-DBFFF	DA000-DAFFF	D9000-D9FFF	D8000-D8FFF	IA32_MTRR_FIX4K_D8000
E7000 E7FFF	E6000-E6FFF	E5000-E5FFF	E4000-E4FFF	E3000-E3FFF	E2000-E2FFF	E1000-E1FFF	E0000-E0FFF	IA32_MTRR_FIX4K_E0000
EF000 EFFFF	EE000-EEFFF	ED000-EDFFF	EC000-ECFFF	EB000-EBFFF	EA000-EAFFF	E9000-E9FFF	E8000-E8FFF	IA32_MTRR_FIX4K_E8000
F7000 F7FFF	F6000-F6FFF	F5000-F5FFF	F4000-F4FFF	F3000-F3FFF	F2000-F2FFF	F1000-F1FFF	F0000-F0FFF	IA32_MTRR_FIX4K_F0000
FF000 FFFFF	FE000-FEFFF	FD000-FDFFF	FC000-FCFFF	FB000-FBFFF	FA000-FAFFF	F9000-F9FFF	F8000-F8FFF	IA32_MTRR_FIX4K_F8000

Figure 13-7 shows flags and fields in these registers. The functions of these flags and fields are:

- **Type field, bits 0 through 7** — Specifies the memory type for the range (see Table 13-8 for the encoding of this field).
- **PhysBase field, bits 12 through (MAXPHYADDR-1)** — Specifies the base address of the address range. This 24-bit value, in the case where MAXPHYADDR is 36 bits, is extended by 12 bits at the low end to form the base address (this automatically aligns the address on a 4-KByte boundary).
- **PhysMask field, bits 12 through (MAXPHYADDR-1)** — Specifies a mask (24 bits if the maximum physical address size is 36 bits, 28 bits if the maximum physical address size is 40 bits). The mask determines the range of the region being mapped, according to the following relationships:
  - Address_Within_Range AND PhysMask = PhysBase AND PhysMask
  - This value is extended by 12 bits at the low end to form the mask value. For more information: see Section 13.11.3, "Example Base and Mask Calculations."
  - The width of the PhysMask field depends on the maximum physical address size supported by the processor.

    CPUID.80000008H reports the maximum physical address size supported by the processor. If CPUID.80000008H is not available, software may assume that the processor supports a 36-bit physical address size (then PhysMask is 24 bits wide and the upper 28 bits of IA32_MTRR_PHYSMASKn are reserved). See the Note below.
- **V (valid) flag, bit 11** — Enables the register pair when set; disables register pair when clear.

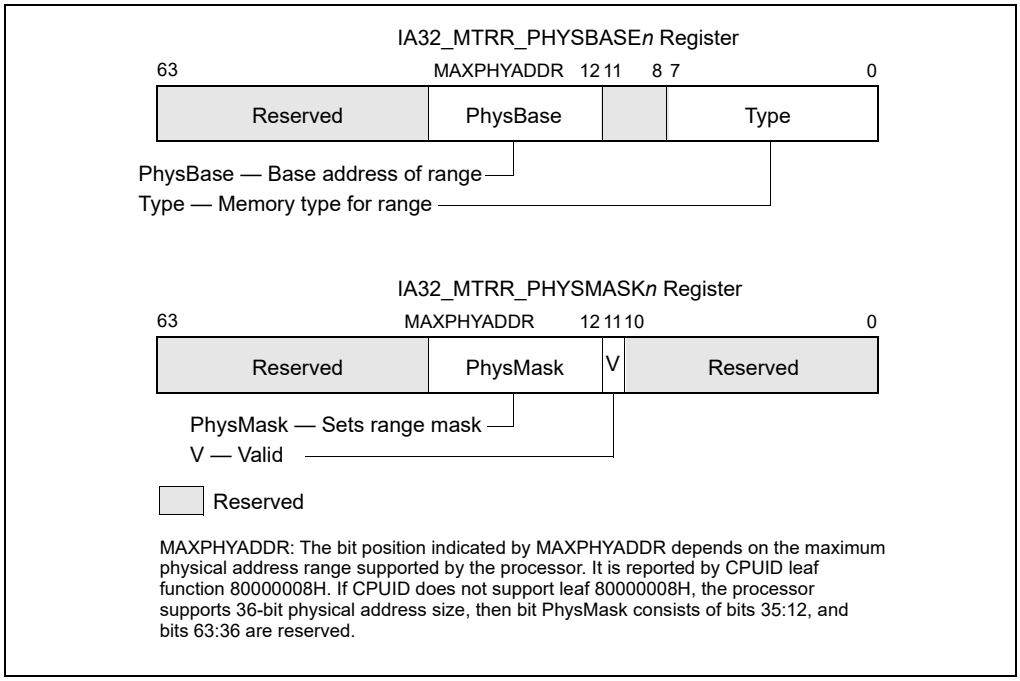


Figure 13-7. IA32_MTRR_PHYSBASE*n* and IA32_MTRR_PHYSMASK*n* Variable-Range Register Pair

All other bits in the IA32_MTRR_PHYSBASE*n* and IA32_MTRR_PHYSMASK*n* registers are reserved; the processor generates a general-protection exception (#GP) if software attempts to write to them.

Some mask values can result in ranges that are not continuous. In such ranges, the area not mapped by the mask value is set to the default memory type, unless some other MTRR specifies a type for that range. Intel does not encourage the use of "discontinuous" ranges.

NOTE

It is possible for software to parse the memory descriptions that BIOS provides by using the ACPI/INT15 e820 interface mechanism. This information then can be used to determine how MTRRs are initialized (for example: allowing the BIOS to define valid memory ranges and the maximum memory range supported by the platform, including the processor).

See Section 13.11.4.1, "MTRR Precedences," for information on overlapping variable MTRR ranges.

13.11.2.4 System-Management Range Register Interface

If IA32_MTRRCAP[bit 11] is set, the processor supports the SMRR interface to restrict access to a specified memory address range used by system-management mode (SMM) software (see Section 33.4.2.1). If the SMRR interface is supported, SMM software is strongly encouraged to use it to protect the SMI code and data stored by SMI handler in the SMRAM region.

The system-management range registers consist of a pair of MSRs (see Figure 13-8). The IA32_SMRR_PHYSBASE MSR defines the base address for the SMRAM memory range and the memory type used to access it in SMM. The IA32_SMRR_PHYSMASK MSR contains a valid bit and a mask that determines the SMRAM address range protected by the SMRR interface. These MSRs may be written only in SMM; an attempt to write them outside of SMM causes a general-protection exception.[1]

Figure 13-8 shows flags and fields in these registers. The functions of these flags and fields are the following:

1. For some processor models, these MSRs can be accessed by RDMSR and WRMSR only if the SMRR interface has been enabled using a model-specific bit in the IA32_FEATURE_CONTROL MSR.

- **Type field, bits 0 through 7** — Specifies the memory type for the range (see Table 13-8 for the encoding of this field).
- **PhysBase field, bits 12 through 31** — Specifies the base address of the address range. The address must be less than 4 GBytes and is automatically aligned on a 4-KByte boundary.
- **PhysMask field, bits 12 through 31** — Specifies a mask that determines the range of the region being mapped, according to the following relationships:
  - Address_Within_Range AND PhysMask = PhysBase AND PhysMask
  - This value is extended by 12 bits at the low end to form the mask value. For more information: see Section 13.11.3, "Example Base and Mask Calculations."
- **V (valid) flag, bit 11** — Enables the register pair when set; disables register pair when clear.

Before attempting to access these SMRR registers, software must test bit 11 in the IA32_MTRRCAP register. If SMRR is not supported, reads from or writes to registers cause general-protection exceptions.

When the valid flag in the IA32_SMRR_PHYSMASK MSR is 1, accesses to the specified address range are treated as follows:

- If the logical processor is in SMM, accesses uses the memory type in the IA32_SMRR_PHYSBASE MSR.
- If the logical processor is not in SMM, write accesses are ignored and read accesses return a fixed value for each byte. The uncacheable memory type (UC) is used in this case.

The above items apply even if the address range specified overlaps with a range specified by the MTRRs.

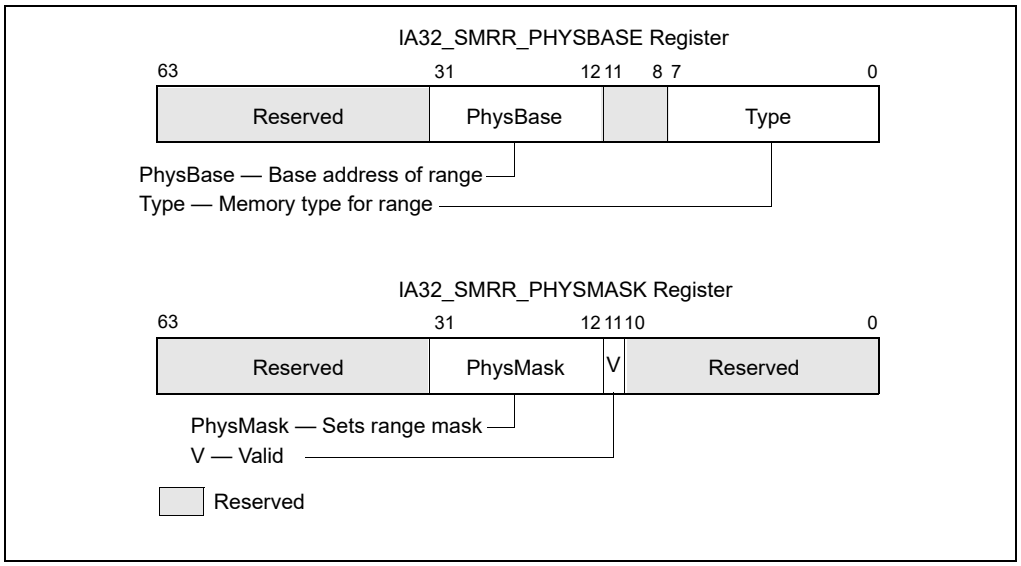


Figure 13-8. IA32_SMRR_PHYSBASE and IA32_SMRR_PHYSMASK SMRR Pair

13.11.3 Example Base and Mask Calculations

The examples in this section apply to processors that support a maximum physical address size of 36 bits. The base and mask values entered in variable-range MTRR pairs are 24-bit values that the processor extends to 36-bits.

For example, to enter a base address of 2 MBytes (200000H) in the IA32_MTRR_PHYSBASE3 register, the 12 least-significant bits are truncated and the value 000200H is entered in the PhysBase field. The same operation must be performed on mask values. For example, to map the address range from 200000H to 3FFFFFH (2 MBytes to 4 MBytes), a mask value of FFFE00000H is required. Again, the 12 least-significant bits of this mask value are truncated, so that the value entered in the PhysMask field of IA32_MTRR_PHYSMASK3 is FFFE00H. This mask is chosen so that when any address in the 200000H to 3FFFFFH range is ANDed with the mask value, it will return the same value as when the base address is ANDed with the mask value (which is 200000H).

To map the address range from 400000H to 7FFFFFH (4 MBytes to 8 MBytes), a base value of 000400H is entered in the PhysBase field and a mask value of FFFC00H is entered in the PhysMask field.

Example 13-2. Setting-Up Memory for a System

Here is an example of setting up the MTRRs for an system. Assume that the system has the following characteristics:

- 96 MBytes of system memory is mapped as write-back memory (WB) for highest system performance.
- A custom 4-MByte I/O card is mapped to uncached memory (UC) at a base address of 64 MBytes. This restriction forces the 96 MBytes of system memory to be addressed from 0 to 64 MBytes and from 68 MBytes to 100 MBytes, leaving a 4-MByte hole for the I/O card.
- An 8-MByte graphics card is mapped to write-combining memory (WC) beginning at address A0000000H.
- The BIOS area from 15 MBytes to 16 MBytes is mapped to UC memory.

The following settings for the MTRRs will yield the proper mapping of the physical address space for this system configuration.

IA32_MTRR_PHYSBASE0 = 0000 0000 0000 0006H
IA32_MTRR_PHYSMASK0 = 0000 000F FC00 0800H
Caches 0-64 MByte as WB cache type.

IA32_MTRR_PHYSBASE1 = 0000 0000 0400 0006H
IA32_MTRR_PHYSMASK1 = 0000 000F FE00 0800H
Caches 64-96 MByte as WB cache type.

IA32_MTRR_PHYSBASE2 = 0000 0000 0600 0006H
IA32_MTRR_PHYSMASK2 = 0000 000F FFC0 0800H
Caches 96-100 MByte as WB cache type.

IA32_MTRR_PHYSBASE3 = 0000 0000 0400 0000H
IA32_MTRR_PHYSMASK3 = 0000 000F FFC0 0800H
Caches 64-68 MByte as UC cache type.

IA32_MTRR_PHYSBASE4 = 0000 0000 00F0 0000H
IA32_MTRR_PHYSMASK4 = 0000 000F FFF0 0800H
Caches 15-16 MByte as UC cache type.

IA32_MTRR_PHYSBASE5 = 0000 0000 A000 0001H
IA32_MTRR_PHYSMASK5 = 0000 000F FF80 0800H
Caches A0000000-A0800000 as WC type.

This MTRR setup uses the ability to overlap any two memory ranges (as long as the ranges are mapped to WB and UC memory types) to minimize the number of MTRR registers that are required to configure the memory environment. This setup also fulfills the requirement that two register pairs are left for operating system usage.

13.11.3.1 Base and Mask Calculations for Greater-Than 36-bit Physical Address Support

For Intel 64 and IA-32 processors that support greater than 36 bits of physical address size, software should query CPUID.80000008H to determine the maximum physical address. See the example.

Example 13-3. Setting-Up Memory for a System with a 40-Bit Address Size

If a processor supports 40-bits of physical address size, then the PhysMask field (in IA32_MTRR_PHYSMASK*n* registers) is 28 bits instead of 24 bits. For this situation, Example 13-2 should be modified as follows:

IA32_MTRR_PHYSBASE0 = 0000 0000 0000 0006H
IA32_MTRR_PHYSMASK0 = 0000 00FF FC00 0800H
Caches 0-64 MByte as WB cache type.

IA32_MTRR_PHYSBASE1 = 0000 0000 0400 0006H
IA32_MTRR_PHYSMASK1 = 0000 00FF FE00 0800H
Caches 64-96 MByte as WB cache type.

IA32_MTRR_PHYSBASE2 = 0000 0000 0600 0006H
IA32_MTRR_PHYSMASK2 = 0000 00FF FFC0 0800H
Caches 96-100 MByte as WB cache type.

IA32_MTRR_PHYSBASE3 = 0000 0000 0400 0000H
IA32_MTRR_PHYSMASK3 = 0000 00FF FFC0 0800H
Caches 64-68 MByte as UC cache type.

IA32_MTRR_PHYSBASE4 = 0000 0000 00F0 0000H
IA32_MTRR_PHYSMASK4 = 0000 00FF FFF0 0800H
Caches 15-16 MByte as UC cache type.

IA32_MTRR_PHYSBASE5 = 0000 0000 A000 0001H
IA32_MTRR_PHYSMASK5 = 0000 00FF FF80 0800H
Caches A0000000-A0800000 as WC type.

13.11.4 Range Size and Alignment Requirement

A range that is to be mapped to a variable-range MTRR must meet the following "power of 2" size and alignment rules:

1. The minimum range size is 4 KBytes and the base address of the range must be on at least a 4-KByte boundary.
2. For ranges greater than 4 KBytes, each range must be of length 2^n and its base address must be aligned on a 2^n boundary, where *n* is a value equal to or greater than 12. The base-address alignment value cannot be less than its length. For example, an 8-KByte range cannot be aligned on a 4-KByte boundary. It must be aligned on at least an 8-KByte boundary.

13.11.4.1 MTRR Precedences

If the MTRRs are not enabled (by setting the E flag in the IA32_MTRR_DEF_TYPE MSR), then all memory accesses are of the UC memory type. If the MTRRs are enabled, then the memory type used for a memory access is determined as follows:

1. If the physical address falls within the first 1 MByte of physical memory and fixed MTRRs are enabled, the processor uses the memory type stored for the appropriate fixed-range MTRR.
2. Otherwise, the processor attempts to match the physical address with a memory type set by the variable-range MTRRs:
   - If one variable memory range matches, the processor uses the memory type stored in the IA32_MTRR_PHYSBASE*n* register for that range.
   - If two or more variable memory ranges match and the memory types are identical, then that memory type is used.
   - If two or more variable memory ranges match and one of the memory types is UC, the UC memory type used.
   - If two or more variable memory ranges match and the memory types are WT and WB, the WT memory type is used.
   - For overlaps not defined by the above rules, processor behavior is undefined.
3. If no fixed or variable memory range matches, the processor uses the default memory type.

13.11.5 MTRR Initialization

On a hardware reset, the P6 and more recent processors clear the valid flags in variable-range MTRRs and clear the E flag in the IA32_MTRR_DEF_TYPE MSR to disable all MTRRs. All other bits in the MTRRs are undefined.

Prior to initializing the MTRRs, software (normally the system BIOS) must initialize all fixed-range and variable-range MTRR register fields to 0. Software can then initialize the MTRRs according to known types of memory, including memory on devices that it auto-configures. Initialization is expected to occur prior to booting the operating system.

See Section 13.11.8, "MTRR Considerations in MP Systems," for information on initializing MTRRs in MP (multiple-processor) systems.

13.11.6 Remapping Memory Types

A system designer may re-map memory types to tune performance or because a future processor may not implement all memory types supported by the Pentium 4, Intel Xeon, and P6 family processors. The following rules support coherent memory-type re-mappings:

1. A memory type should not be mapped into another memory type that has a weaker memory ordering model. For example, the uncacheable type cannot be mapped into any other type, and the write-back, write-through, and write-protected types cannot be mapped into the weakly ordered write-combining type.
2. A memory type that does not delay writes should not be mapped into a memory type that does delay writes, because applications of such a memory type may rely on its write-through behavior. Accordingly, the write-back type cannot be mapped into the write-through type.
3. A memory type that views write data as not necessarily stored and read back by a subsequent read, such as the write-protected type, can only be mapped to another type with the same behavior (and there are no others for the Pentium 4, Intel Xeon, and P6 family processors) or to the uncacheable type.

In many specific cases, a system designer can have additional information about how a memory type is used, allowing additional mappings. For example, write-through memory with no associated write side effects can be mapped into write-back memory.

13.11.7 MTRR Maintenance Programming Interface

The operating system maintains the MTRRs after booting and sets up or changes the memory types for memory-mapped devices. The operating system should provide a driver and application programming interface (API) to access and set the MTRRs. The function calls MemTypeGet() and MemTypeSet() define this interface.

13.11.7.1 MemTypeGet() Function

The MemTypeGet() function returns the memory type of the physical memory range specified by the parameters base and size. The base address is the starting physical address and the size is the number of bytes for the memory range. The function automatically aligns the base address and size to 4-KByte boundaries. Pseudocode for the MemTypeGet() function is given in Example 13-4.

Example 13-4. MemTypeGet() Pseudocode

```
#define MIXED_TYPES -1     /* 0 < MIXED_TYPES || MIXED_TYPES > 256 */

IF CPU_FEATURES.MTRR /* processor supports MTRRs */
    THEN
        Align BASE and SIZE to 4-KByte boundary;
        IF (BASE + SIZE) wrap physical-address space
            THEN return INVALID;
        FI;
        IF MTRRdefType.E = 0
            THEN return UC;
        FI;
        FirstType := Get4KMemType (BASE);
        /* Obtains memory type for first 4-KByte range. */
        /* See Get4KMemType (4KByteRange) in Example 13-5. */
        FOR each additional 4-KByte range specified in SIZE
            NextType := Get4KMemType (4KByteRange);
            IF NextType != FirstType
                THEN return Mixed_Types;
            FI;
        ROF;
        return FirstType;
    ELSE return UNSUPPORTED;
FI;
```

If the processor does not support MTRRs, the function returns UNSUPPORTED. If the MTRRs are not enabled, then the UC memory type is returned. If more than one memory type corresponds to the specified range, a status of MIXED_TYPES is returned. Otherwise, the memory type defined for the range (UC, WC, WT, WB, or WP) is returned.

The pseudocode for the Get4KMemType() function in Example 13-5 obtains the memory type for a single 4-KByte range at a given physical address. The sample code determines whether an PHY_ADDRESS falls within a fixed range by comparing the address with the known fixed ranges: 0 to 7FFFFH (64-KByte regions), 80000H to BFFFFH (16-KByte regions), and C0000H to FFFFFH (4-KByte regions). If an address falls within one of these ranges, the appropriate bits within one of its MTRRs determine the memory type.

Example 13-5. Get4KMemType() Pseudocode

```
IF IA32_MTRRCAP.FIX AND MTRRdefType.FE /* fixed registers enabled */
   THEN IF PHY_ADDRESS is within a fixed range
        return IA32_MTRR_FIX.Type;
FI;
FOR each variable-range MTRR in IA32_MTRRCAP.VCNT
    IF IA32_MTRR_PHYSMASK.V = 0
        THEN continue;
    FI;
    IF (PHY_ADDRESS AND IA32_MTRR_PHYSMASK.Mask) =
            (IA32_MTRR_PHYSBASE.Base
            AND IA32_MTRR_PHYSMASK.Mask)
        THEN
            return IA32_MTRR_PHYSBASE.Type;
    FI;
ROF;
return MTRRdefType.Type;
```

13.11.7.2 MemTypeSet() Function

The MemTypeSet() function in Example 13-6 sets a MTRR for the physical memory range specified by the parameters base and size to the type specified by type. The base address and size are multiples of 4 KBytes and the size is not 0.

Example 13-6. MemTypeSet Pseudocode

```
IF CPU_FEATURES.MTRR (* processor supports MTRRs *)
   THEN
      IF BASE and SIZE are not 4-KByte aligned or size is 0
         THEN return INVALID;
      FI;
      IF (BASE + SIZE) wrap 4-GByte address space
         THEN return INVALID;
      FI;
      IF TYPE is invalid for Pentium 4, Intel Xeon, and P6 family
      processors
         THEN return UNSUPPORTED;
      FI;
      IF TYPE is WC and not supported
         THEN return UNSUPPORTED;
      FI;
      IF IA32_MTRRCAP.FIX is set AND range can be mapped using a
      fixed-range MTRR
         THEN
            pre_mtrr_change();
            update affected MTRR;
            post_mtrr_change();
      FI;

   ELSE (* try to map using a variable MTRR pair *)
      IF IA32_MTRRCAP.VCNT = 0
         THEN return UNSUPPORTED;
      FI;
      IF conflicts with current variable ranges
         THEN return RANGE_OVERLAP;
      FI;
      IF no MTRRs available
         THEN return VAR_NOT_AVAILABLE;
      FI;
      IF BASE and SIZE do not meet the power of 2 requirements for
      variable MTRRs
         THEN return INVALID_VAR_REQUEST;
      FI;
      pre_mtrr_change();
      Update affected MTRRs;
      post_mtrr_change();
FI;

pre_mtrr_change()
   BEGIN
      disable interrupts;
      Save current value of CR4;
      disable and flush caches;
```

```
      flush TLBs;
      disable MTRRs;
      IF multiprocessing
         THEN maintain consistency through IPIs;
      FI;
   END
post_mtrr_change()
   BEGIN
      flush caches and TLBs;
      enable MTRRs;
      enable caches;
      restore value of CR4;
      enable interrupts;
   END
```

The physical address to variable range mapping algorithm in the MemTypeSet function detects conflicts with current variable range registers by cycling through them and determining whether the physical address in question matches any of the current ranges. During this scan, the algorithm can detect whether any current variable ranges overlap and can be concatenated into a single range.

The pre_mtrr_change() function disables interrupts prior to changing the MTRRs, to avoid executing code with a partially valid MTRR setup. The algorithm disables caching by setting the CD flag and clearing the NW flag in control register CR0. The caches are invalidated using the WBINVD instruction. The algorithm flushes all TLB entries either by clearing the page-global enable (PGE) flag in control register CR4 (if PGE was already set) or by updating control register CR3 (if PGE was already clear). Finally, it disables MTRRs by clearing the E flag in the IA32_MTRR_DEF_TYPE MSR.

After the memory type is updated, the post_mtrr_change() function re-enables the MTRRs and again invalidates the caches and TLBs. This second invalidation is required because of the processor's aggressive prefetch of both instructions and data. The algorithm restores interrupts and re-enables caching by setting the CD flag.

An operating system can batch multiple MTRR updates so that only a single pair of cache invalidations occur.

13.11.8 MTRR Considerations in MP Systems

In MP (multiple-processor) systems, the operating systems must maintain MTRR consistency between all the processors in the system. The Pentium 4, Intel Xeon, and P6 family processors provide no hardware support to maintain this consistency. In general, all processors must have the same MTRR values.

This requirement implies that when the operating system initializes an MP system, it must load the MTRRs of the boot processor while the E flag in register MTRRdefType is 0. The operating system then directs other processors to load their MTRRs with the same memory map. After all the processors have loaded their MTRRs, the operating system signals them to enable their MTRRs. Barrier synchronization is used to prevent further memory accesses until all processors indicate that the MTRRs are enabled. This synchronization is likely to be a shoot-down style algorithm, with shared variables and interprocessor interrupts.

Any change to the value of the MTRRs in an MP system requires the operating system to repeat the loading and enabling process to maintain consistency, using the following procedure:

1. Broadcast to all processors to execute the following code sequence.
2. Disable interrupts.
3. Wait for all processors to reach this point.
4. Enter the no-fill cache mode. (Set the CD flag in control register CR0 to 1 and the NW flag to 0.)
5. Flush all caches using the WBINVD instructions. On a processor that supports self-snooping (enumerating CPUID.01H:EDX.SS[bit 27] as 1), this step may be unnecessary. However, if there are changes for which self-snooping behavior would be problematic (e.g., changing the memory type of a cache line from WB to UC for memory-mapped I/O), execution of WBINVD would still be required.

6. If either the PGE or PCIDE flag is set in control register CR4, flush all TLBs by clearing one or both of these flags.
7. If the PGE and PCIDE flags are both clear in control register CR4, flush all TLBs by executing a MOV from control register CR3 to another register and then a MOV from that register back to CR3.
8. Disable all range registers (by clearing the E flag in register MTRRdefType). If only variable ranges are being modified, software may clear the valid bits for the affected register pairs instead.
9. Update the MTRRs.
10. Enable all range registers (by setting the E flag in register MTRRdefType). If only variable-range registers were modified and their individual valid bits were cleared, then set the valid bits for the affected ranges instead.
11. Flush all caches and all TLBs a second time. (The TLB flush is required for Pentium 4, Intel Xeon, and P6 family processors. Executing the WBINVD instruction is not needed when using Pentium 4, Intel Xeon, and P6 family processors, but it may be needed in future systems.)
12. Enter the normal cache mode to re-enable caching. (Set the CD and NW flags in control register CR0 to 0.)
13. Restore the values of the PGE and/or PCIDE flags in control register CR4, if cleared in Step 6 (above).
14. Wait for all processors to reach this point.
15. Enable interrupts.

13.11.9 Large Page Size Considerations

The MTRRs provide memory typing for a limited number of regions that have a 4 KByte granularity (the same granularity as 4-KByte pages). The memory type for a given page is cached in the processor's TLBs. When using large pages (2 MBytes, 4 MBytes, or 1 GBytes), a single page-table entry covers multiple 4-KByte granules, each with a single memory type. Because the memory type for a large page is cached in the TLB, the processor can behave in an undefined manner if a large page is mapped to a region of memory that MTRRs have mapped with multiple memory types.

Undefined behavior can be avoided by ensuring that all MTRR memory-type ranges within a large page are of the same type. If a large page maps to a region of memory containing different MTRR-defined memory types, the PCD and PWT flags in the page-table entry should be set for the most conservative memory type for that range. For example, a large page used for memory mapped I/O and regular memory is mapped as UC memory. Alternatively, the operating system can map the region using multiple 4-KByte pages each with its own memory type.

The requirement that all 4-KByte ranges in a large page are of the same memory type implies that large pages with different memory types may suffer a performance penalty, since they must be marked with the lowest common denominator memory type. The same consideration apply to 1 GByte pages, each of which may consist of multiple 2-Mbyte ranges.

The Pentium 4, Intel Xeon, and P6 family processors provide special support for the physical memory range from 0 to 4 MBytes, which is potentially mapped by both the fixed and variable MTRRs. This support is invoked when a Pentium 4, Intel Xeon, or P6 family processor detects a large page overlapping the first 1 MByte of this memory range with a memory type that conflicts with the fixed MTRRs. Here, the processor maps the memory range as multiple 4-KByte pages within the TLB. This operation ensures correct behavior at the cost of performance. To avoid this performance penalty, operating-system software should reserve the large page option for regions of memory at addresses greater than or equal to 4 MBytes.

13.12 PAGE ATTRIBUTE TABLE (PAT)

The Page Attribute Table (PAT) extends the IA-32 architecture's page-table format to allow memory types to be assigned to regions of physical memory based on linear address mappings. The PAT is a companion feature to the MTRRs; that is, the MTRRs allow mapping of memory types to regions of the physical address space, where the PAT allows mapping of memory types to pages within the linear address space. The MTRRs are useful for statically describing memory types for physical ranges, and are typically set up by the system BIOS. The PAT extends the functions of the PCD and PWT bits in page tables to allow all five of the memory types that can be assigned with the MTRRs (plus one additional memory type) to also be assigned dynamically to pages of the linear address space.

The PAT was introduced to IA-32 architecture on the Pentium III processor. It is also available in the Pentium 4 and Intel Xeon processors.

13.12.1 Detecting Support for the PAT Feature

An operating system or executive can detect the availability of the PAT by executing the CPUID instruction with a value of 1 in the EAX register. Support for the PAT is indicated by the PAT flag (bit 16 of the values returned to EDX register). If the PAT is supported, the operating system or executive can use the IA32_PAT MSR to program the PAT. When memory types have been assigned to entries in the PAT, software can then use of the PAT-index bit (PAT) in the page-table and page-directory entries along with the PCD and PWT bits to assign memory types from the PAT to individual pages.

Note that there is no separate flag or control bit in any of the control registers that enables the PAT. The PAT is always enabled on all processors that support it, and the table lookup always occurs whenever paging is enabled, in all paging modes.

13.12.2 IA32_PAT MSR

The IA32_PAT MSR is located at MSR address 277H (see Chapter 2, "Model-Specific Registers (MSRs)," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4). Figure 13-9. shows the format of the 64-bit IA32_PAT MSR.

The IA32_PAT MSR contains eight page attribute fields: PA0 through PA7. The three low-order bits of each field are used to specify a memory type. The five high-order bits of each field are reserved, and must be set to all 0s. Each of the eight page attribute fields can contain any of the memory type encodings specified in Table 13-10.

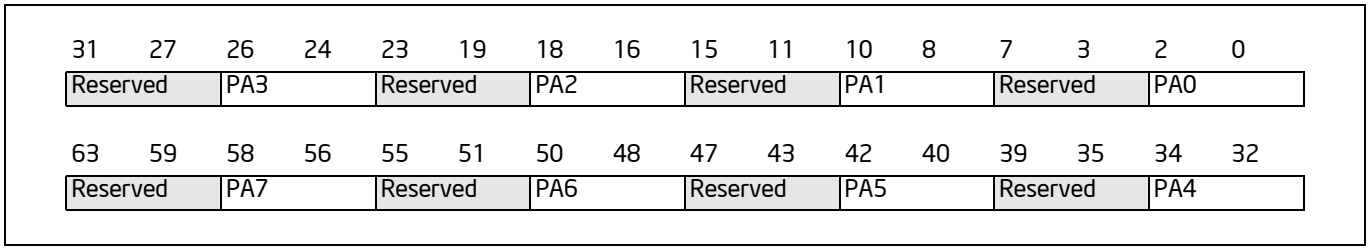


Figure 13-9. IA32_PAT MSR

Note that for the P6 family processors, the IA32_PAT MSR is named the PAT MSR.

Table 13-10. Memory Types That Can Be Encoded With PAT

Encoding	Mnemonic
00H	Uncacheable (UC)
01H	Write Combining (WC)
02H	Reserved*
03H	Reserved*
04H	Write Through (WT)
05H	Write Protected (WP)
06H	Write Back (WB)
07H	Uncached (UC-)
08H - FFH	Reserved*

NOTE:

* Using these encodings will result in a general-protection exception (#GP).

13.12.3 Selecting a Memory Type from the PAT

To select a memory type for a page from the PAT, a 3-bit index made up of the PAT, PCD, and PWT bits must be encoded in the page-table or page-directory entry for the page. Table 13-11 shows the possible encodings of the PAT, PCD, and PWT bits and the PAT entry selected with each encoding. The PAT bit is bit 7 in page-table entries that point to 4-KByte pages and bit 12 in paging-structure entries that point to larger pages. The PCD and PWT bits are bits 4 and 3, respectively, in paging-structure entries that point to pages of any size.

The PAT entry selected for a page is used in conjunction with the MTRR setting for the region of physical memory in which the page is mapped to determine the effective memory type for the page, as shown in Table 13-7.

Table 13-11. Selection of PAT Entries with PAT, PCD, and PWT Flags

PAT	PCD	PWT	PAT Entry
0	0	0	PAT0
0	0	1	PAT1
0	1	0	PAT2
0	1	1	PAT3
1	0	0	PAT4
1	0	1	PAT5
1	1	0	PAT6
1	1	1	PAT7

13.12.4 Programming the PAT

Table 13-12 shows the default setting for each PAT entry following a power up or reset of the processor. The setting remain unchanged following a soft reset (INIT reset).

Table 13-12. Memory Type Setting of PAT Entries Following a Power-up or Reset

PAT Entry	Memory Type Following Power-up or Reset
PAT0	WB
PAT1	WT
PAT2	UC-
PAT3	UC
PAT4	WB
PAT5	WT
PAT6	UC-
PAT7	UC

The values in all the entries of the PAT can be changed by writing to the IA32_PAT MSR using the WRMSR instruction. The IA32_PAT MSR is read and write accessible (use of the RDMSR and WRMSR instructions, respectively) to software operating at a CPL of 0. Table 13-10 shows the allowable encoding of the entries in the PAT. Attempting to write an undefined memory type encoding into the PAT causes a general-protection (#GP) exception to be generated.

The operating system (OS) is responsible for ensuring that changes to a PAT entry occur in a manner that maintains the consistency of the processor caches and translation lookaside buffers (TLB). It requires the OS to invalidate all affected TLB entries (including global entries) and all entries in all paging-structure caches. It may also require flushing of the processor caches in certain situations. This can be accomplished in various ways, including the sequence below or by following the procedure specified in Section 13.11.8, “MTRR Considerations in MP Systems.” (See Section 5.10.4, “Invalidation of TLBs and Paging-Structure Caches” for additional background information.) Also note that in a multi-processor environment, it is the software's responsibility to resolve differences in conflicting memory types across logical processors that may arise from changes to the PAT (e.g., if two

logical processors map a linear address to the same physical address but have PATs that specify a different memory type for that physical address).

Example of a sequence to invalidate the processor TLBs and caches (if necessary):

1. If the PCIDE or PGE flag is set in CR4, flush TLBs by clearing one of those flags (then restore the flag via a subsequent CR4 write).

   Otherwise, flush TLBs by executing a MOV from control register CR3 to another register and then a MOV from that register back to CR3.

2. In the case that there are changes to memory-type mappings for which cache self-snooping behavior would be problematic given the existing mappings (e.g., changing a cache line's memory type from WB to UC to be used for memory-mapped I/O), then cache flushing is also required. This can be done by executing CLFLUSH operations for all affected cache lines or by executing the WBINVD instruction (recommended only if there are a large number of affected mappings or if it is unknown which mappings are affected).

The PAT allows any memory type to be specified in the page tables, and therefore it is possible to have a single physical page mapped to two or more different linear addresses, each with different memory types. Intel does not support this practice because it may lead to undefined operations that can result in a system failure. In particular, a WC page must never be aliased to a cacheable page because WC writes may not check the processor caches.

When remapping a page that was previously mapped as a cacheable memory type to a WC page, an operating system can avoid this type of aliasing by doing the following:

1. Remove the previous mapping to a cacheable memory type in the page tables; that is, make them not present.
2. Flush the TLBs of processors that may have used the mapping, even speculatively.
3. Create a new mapping to the same physical address with a new memory type, for instance, WC.
4. Flush the caches on all processors that may have used the mapping previously. Note on processors that support self-snooping, CPUID feature flag bit 27, this step is unnecessary.

Operating systems that use a page directory as a page table (to map large pages) and enable page size extensions must carefully scrutinize the use of the PAT index bit for the 4-KByte page-table entries. The PAT index bit for a page-table entry (bit 7) corresponds to the page size bit in a page-directory entry. Therefore, the operating system can only use PAT entries PA0 through PA3 when setting the caching type for a page table that is also used as a page directory. If the operating system attempts to use PAT entries PA4 through PA7 when using this memory as a page table, it effectively sets the PS bit for the access to this memory as a page directory.

For compatibility with earlier IA-32 processors that do not support the PAT, care should be taken in selecting the encodings for entries in the PAT (see Section 13.12.5, “PAT Compatibility with Earlier IA-32 Processors”).

13.12.5 PAT Compatibility with Earlier IA-32 Processors

For IA-32 processors that support the PAT, the IA32_PAT MSR is always active. That is, the PCD and PWT bits in page-table entries and in page-directory entries (that point to pages) are always select a memory type for a page indirectly by selecting an entry in the PAT. They never select the memory type for a page directly as they do in earlier IA-32 processors that do not implement the PAT (see Table 13-6).

To allow compatibility for code written to run on earlier IA-32 processor that do not support the PAT, the PAT mechanism has been designed to allow backward compatibility to earlier processors. This compatibility is provided through the ordering of the PAT, PCD, and PWT bits in the 3-bit PAT entry index. For processors that do not implement the PAT, the PAT index bit (bit 7 in the page-table entries and bit 12 in the page-directory entries) is reserved and set to 0. With the PAT bit reserved, only the first four entries of the PAT can be selected with the PCD and PWT bits. At power-up or reset (see Table 13-12), these first four entries are encoded to select the same memory types as the PCD and PWT bits would normally select directly in an IA-32 processor that does not implement the PAT. So, if encodings of the first four entries in the PAT are left unchanged following a power-up or reset, code written to run on earlier IA-32 processors that do not implement the PAT will run correctly on IA-32 processors that do implement the PAT.

CHAPTER 14
INTEL® MMX™ TECHNOLOGY SYSTEM PROGRAMMING

This chapter describes those features of the Intel® MMX™ technology that must be considered when designing or enhancing an operating system to support MMX technology. It covers MMX instruction set emulation, the MMX state, aliasing of MMX registers, saving MMX state, task and context switching considerations, exception handling, and debugging.

14.1 EMULATION OF THE MMX INSTRUCTION SET

The IA-32 or Intel 64 architecture does not support emulation of the MMX instructions, as it does for x87 FPU instructions. The EM flag in control register CR0 (provided to invoke emulation of x87 FPU instructions) cannot be used for MMX instruction emulation. If an MMX instruction is executed when the EM flag is set, an invalid opcode exception (UD#) is generated. Table 14-1 shows the interaction of the EM, MP, and TS flags in control register CR0 when executing MMX instructions.

Table 14-1. Action Taken By MMX Instructions for Different Combinations of EM, MP, and TS

CR0 Flags			Action
EM	MP*	TS	
0	1	0	Execute.
0	1	1	#NM exception.
1	1	0	#UD exception.
1	1	1	#UD exception.

NOTE:

* For processors that support the MMX instructions, the MP flag should be set.

14.2 THE MMX STATE AND MMX REGISTER ALIASING

The MMX state consists of eight 64-bit registers (MM0 through MM7). These registers are aliased to the low 64-bits (bits 0 through 63) of floating-point registers R0 through R7 (see Figure 14-1). Note that the MMX registers are mapped to the physical locations of the floating-point registers (R0 through R7), not to the relative locations of the registers in the floating-point register stack (ST0 through ST7). As a result, the MMX register mapping is fixed and is not affected by value in the Top Of Stack (TOS) field in the floating-point status word (bits 11 through 13).

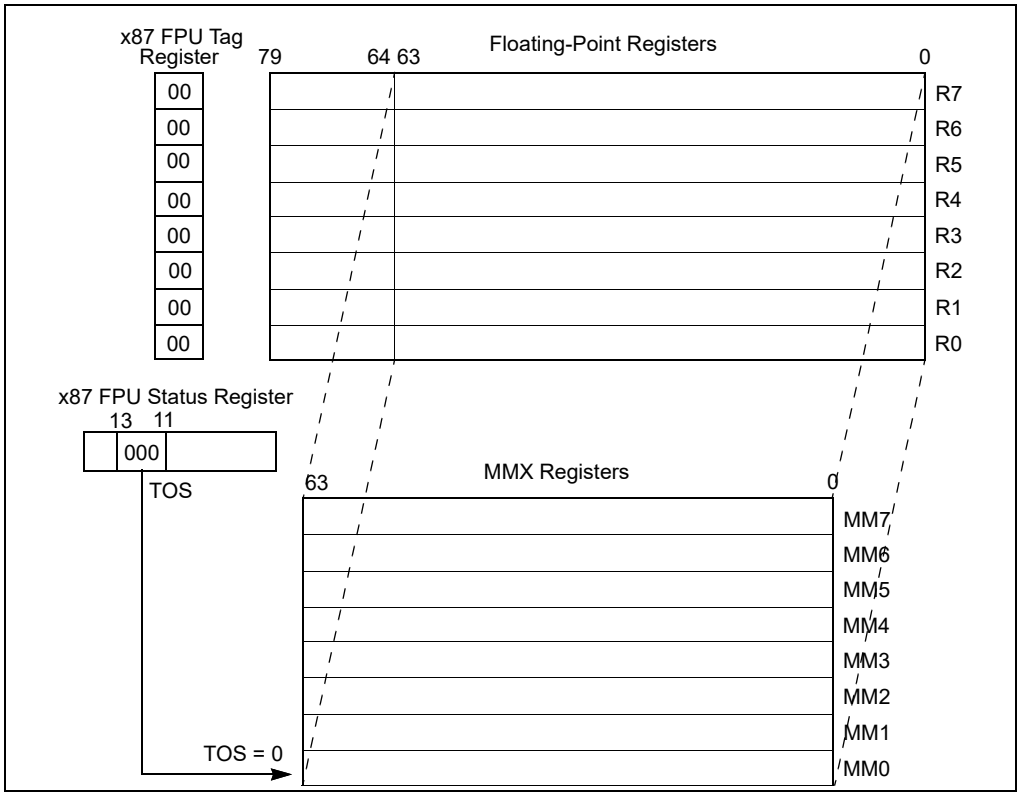


Figure 14-1. Mapping of MMX Registers to Floating-Point Registers

When a value is written into an MMX register using an MMX instruction, the value also appears in the corresponding floating-point register in bits 0 through 63. Likewise, when a floating-point value written into a floating-point register by a x87 FPU, the low 64 bits of that value also appears in a the corresponding MMX register.

The execution of MMX instructions have several side effects on the x87 FPU state contained in the floating-point registers, the x87 FPU tag word, and the x87 FPU status word. These side effects are as follows:

- When an MMX instruction writes a value into an MMX register, at the same time, bits 64 through 79 of the corresponding floating-point register are set to all 1s.
- When an MMX instruction (other than the EMMS instruction) is executed, each of the tag fields in the x87 FPU tag word is set to 00B (valid). (See also Section 14.2.1, "Effect of MMX, x87 FPU, FXSAVE, and FXRSTOR Instructions on the x87 FPU Tag Word.")
- When the EMMS instruction is executed, each tag field in the x87 FPU tag word is set to 11B (empty).
- Each time an MMX instruction is executed, the TOS value is set to 000B.

Execution of MMX instructions does not affect the other bits in the x87 FPU status word (bits 0 through 10 and bits 14 and 15) or the contents of the other x87 FPU registers that comprise the x87 FPU state (the x87 FPU control word, instruction pointer, data pointer, or opcode registers).

Table 14-2 summarizes the effects of the MMX instructions on the x87 FPU state.

Table 14-2. Effects of MMX Instructions on x87 FPU State

MMX Instruction Type	x87 FPU Tag Word	TOS Field of x87 FPU Status Word	Other x87 FPU Registers	Bits 64 Through 79 of x87 FPU Data Registers	Bits 0 Through 63 of x87 FPU Data Registers
Read from MMX register	All tags set to 00B (Valid)	000B	Unchanged	Unchanged	Unchanged
Write to MMX register	All tags set to 00B (Valid)	000B	Unchanged	Set to all 1s	Overwritten with MMX data
EMMS	All fields set to 11B (Empty)	000B	Unchanged	Unchanged	Unchanged

14.2.1 Effect of MMX, x87 FPU, FXSAVE, and FXRSTOR Instructions on the x87 FPU Tag Word

Table 14-3 summarizes the effect of MMX and x87 FPU instructions and the FXSAVE and FXRSTOR instructions on the tags in the x87 FPU tag word and the corresponding tags in an image of the tag word stored in memory.

The values in the fields of the x87 FPU tag word do not affect the contents of the MMX registers or the execution of MMX instructions. However, the MMX instructions do modify the contents of the x87 FPU tag word, as is described in Section 14.2, "The MMX State and MMX Register Aliasing." These modifications may affect the operation of the x87 FPU when executing x87 FPU instructions, if the x87 FPU state is not initialized or restored prior to beginning x87 FPU instruction execution.

Note that the FSAVE, FXSAVE, and FSTENV instructions (which save x87 FPU state information) read the x87 FPU tag register and contents of each of the floating-point registers, determine the actual tag values for each register (empty, nonzero, zero, or special), and store the updated tag word in memory. After executing these instructions, all the tags in the x87 FPU tag word are set to empty (11B). Likewise, the EMMS instruction clears MMX state from the MMX/floating-point registers by setting all the tags in the x87 FPU tag word to 11B.

Table 14-3. Effect of the MMX, x87 FPU, and FXSAVE/FXRSTOR Instructions on the x87 FPU Tag Word

Instruction Type	Instruction	x87 FPU Tag Word	Image of x87 FPU Tag Word Stored in Memory
MMX	All (except EMMS)	All tags are set to 00B (valid).	Not affected.
MMX	EMMS	All tags are set to 11B (empty).	Not affected.
x87 FPU	All (except FSAVE, FSTENV, FRSTOR, FLDENV)	Tag for modified floating-point register is set to 00B or 11B.	Not affected.
x87 FPU and FXSAVE	FSAVE, FSTENV, FXSAVE	Tags and register values are read and interpreted; then all tags are set to 11B.	Tags are set according to the actual values in the floating-point registers; that is, empty registers are marked 11B and valid registers are marked 00B (nonzero), 01B (zero), or 10B (special).
x87 FPU and FXRSTOR	FRSTOR, FLDENV, FXRSTOR	All tags marked 11B in memory are set to 11B; all other tags are set according to the value in the corresponding floating-point register: 00B (nonzero), 01B (zero), or 10B (special).	Tags are read and interpreted, but not modified.

14.3 SAVING AND RESTORING THE MMX STATE AND REGISTERS

Because the MMX registers are aliased to the x87 FPU data registers, the MMX state can be saved to memory and restored from memory as follows:

- Execute an FSAVE, FNSAVE, or FXSAVE instruction to save the MMX state to memory. (The FXSAVE instruction also saves the state of the XMM and MXCSR registers.)
- Execute an FRSTOR or FXRSTOR instruction to restore the MMX state from memory. (The FXRSTOR instruction also restores the state of the XMM and MXCSR registers.)

The save and restore methods described above are required for operating systems (see Section 14.4, "Saving MMX State on Task or Context Switches"). Applications can in some cases save and restore only the MMX registers in the following way:

- Execute eight MOVQ instructions to save the contents of the MMX0 through MMX7 registers to memory. An EMMS instruction may then (optionally) be executed to clear the MMX state in the x87 FPU.
- Execute eight MOVQ instructions to read the saved contents of MMX registers from memory into the MMX0 through MMX7 registers.

NOTE

The IA-32 architecture does not support scanning the x87 FPU tag word and then only saving valid entries.

14.4 SAVING MMX STATE ON TASK OR CONTEXT SWITCHES

When switching from one task or context to another, it is often necessary to save the MMX state. As a general rule, if the existing task switching code for an operating system includes facilities for saving the state of the x87 FPU, these facilities can also be relied upon to save the MMX state, without rewriting the task switch code. This reliance is possible because the MMX state is aliased to the x87 FPU state (see Section 14.2, "The MMX State and MMX Register Aliasing").

With the introduction of the FXSAVE and FXRSTOR instructions and of SSE/SSE2/SSE3/SSSE3 extensions, it is possible (and more efficient) to create state saving facilities in the operating system or executive that save the x87 FPU/MMX/SSE/SSE2/SSE3/SSSE3 state in one operation. Section 15.4, "Designing OS Facilities for Saving x87 FPU, SSE, AND EXTENDED States on Task or Context Switches," describes how to design such facilities. The techniques describes in this section can be adapted to saving only the MMX and x87 FPU state if needed.

14.5 EXCEPTIONS THAT CAN OCCUR WHEN EXECUTING MMX INSTRUCTIONS

MMX instructions do not generate x87 FPU floating-point exceptions, nor do they affect the processor's status flags in the EFLAGS register or the x87 FPU status word. The following exceptions can be generated during the execution of an MMX instruction:

- Exceptions during memory accesses:
  - — Stack-segment fault (#SS).
  - — General protection (#GP).
  - — Page fault (#PF).
  - — Alignment check (#AC), if alignment checking is enabled.
- System exceptions:
  - — Invalid Opcode (#UD), if the EM flag in control register CR0 is set when an MMX instruction is executed (see Section 14.1, "Emulation of the MMX Instruction Set").
  - — Device not available (#NM), if an MMX instruction is executed when the TS flag in control register CR0 is set. (See Section 15.4.1, "Using the TS Flag to Control the Saving of the x87 FPU and SSE State.")
- Floating-point error (#MF). (See Section 14.5.1, "Effect of MMX Instructions on Pending x87 Floating-Point Exceptions.")
- Other exceptions can occur indirectly due to the faulty execution of the exception handlers for the above exceptions.

14.5.1 Effect of MMX Instructions on Pending x87 Floating-Point Exceptions

If an x87 FPU floating-point exception is pending and the processor encounters an MMX instruction, the processor generates a x87 FPU floating-point error (#MF) prior to executing the MMX instruction, to allow the pending exception to be handled by the x87 FPU floating-point error exception handler. While this exception handler is executing, the x87 FPU state is maintained and is visible to the handler. Upon returning from the exception handler, the MMX instruction is executed, which will alter the x87 FPU state, as described in Section 14.2, "The MMX State and MMX Register Aliasing."

14.6 DEBUGGING MMX CODE

The debug facilities operate in the same manner when executing MMX instructions as when executing other IA-32 or Intel 64 architecture instructions.

To correctly interpret the contents of the MMX or x87 FPU registers from the FSAVE/FNSAVE or FXSAVE image in memory, a debugger needs to take account of the relationship between the x87 FPU register's logical locations relative to TOS and the MMX register's physical locations.

In the x87 FPU context, ST*n* refers to an x87 FPU register at location *n* relative to the TOS. However, the tags in the x87 FPU tag word are associated with the physical locations of the x87 FPU registers (R0 through R7). The MMX registers always refer to the physical locations of the registers (with MM0 through MM7 being mapped to R0 through R7). Figure 14-2 shows this relationship. Here, the inner circle refers to the physical location of the x87 FPU and MMX registers. The outer circle refers to the x87 FPU register's relative location to the current TOS.

When the TOS equals 0 (case A in Figure 14-2), ST0 points to the physical location R0 on the floating-point stack. MM0 maps to ST0, MM1 maps to ST1, and so on.

When the TOS equals 2 (case B in Figure 14-2), ST0 points to the physical location R2. MM0 maps to ST6, MM1 maps to ST7, MM2 maps to ST0, and so on.

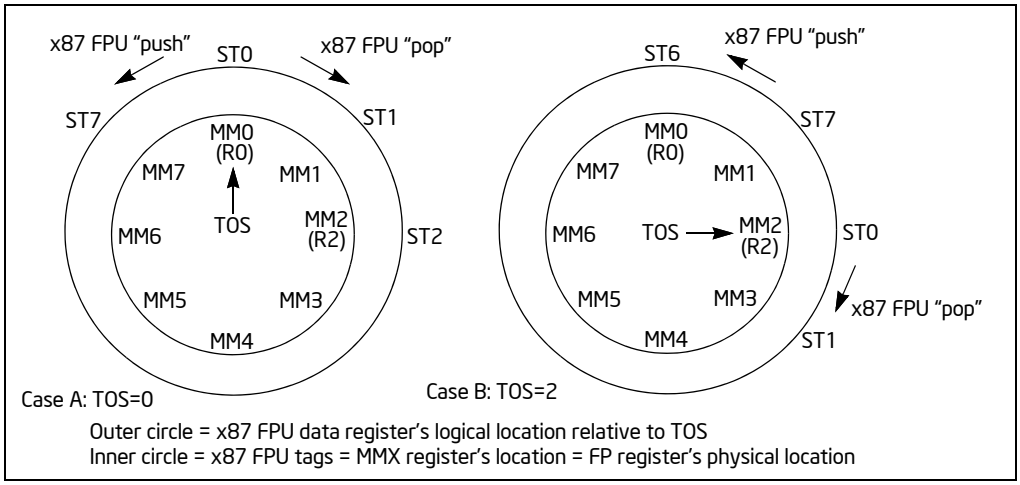


Figure 14-2. Mapping of MMX Registers to x87 FPU Data Register Stack

CHAPTER 15
SYSTEM PROGRAMMING FOR INSTRUCTION SET EXTENSIONS AND PROCESSOR EXTENDED STATES

This chapter describes system programming features for instruction set extensions operating on the processor state extension known as the SSE state (XMM registers, MXCSR) and for other processor extended states. Instruction set extensions operating on the SSE state include the streaming SIMD extensions (SSE), streaming SIMD extensions 2 (SSE2), streaming SIMD extensions 3 (SSE3), Supplemental SSE3 (SSSE3), and SSE4. Collectively, these are called **SSE extensions**[1] and the corresponding instructions are called **SSE instructions**. FXSAVE/FXRSTOR instructions can be used save/restore SSE state along with FP state. See Section 10.5 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for information about FXSAVE and FXRSTOR.

Sections 15.1 through 15.4 cover system programming requirements to enable the SSE extensions, providing operating system or executive support for the SSE extensions, SIMD floating-point exceptions, exception handling, and task (context) switching. These sections primarily discuss use of FXSAVE/FXRSTOR to save/restore SSE state.

XSAVE feature set refers to extensions to the Intel architecture that will allow system executives to implement support for multiple processor extended states along with FP/SSE states that may be introduced over time without requiring the system executive to be modified each time a new processor state extension is introduced. XSAVE feature set provide mechanisms to enumerate the supported extended states, enable some or all of them for software use, instructions to save/restore the states and enumerate the layout of the states when saved to memory. XSAVE/XRSTOR instructions are part of the XSAVE feature set. These instructions are introduced after the introduction of FP/SSE states but can be used to manage legacy FP/SSE state along with processor extended states. See Chapter 13 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for information about XSAVE feature set.

System programming for managing processor extended states is described in sections 15.5 through 15.6. XSAVE feature set is designed to be compatible with FXSAVE/FXRSTOR and hence much of the material through sections 15.1 to 15.4 related to SSE state also applies to XSAVE feature set with the exception of enumeration and saving/restoring state.

XSAVE Compaction is an XSAVE feature that allows operating systems to allocate space for only the states saved to conserve memory usage. A new instruction called XSAVEC is introduced to save extended states in compacted format and XRSTOR instruction is enhanced to comprehend compacted format. System programming for managing processor extended states in compacted format is also described in section 15.5.

Supervisor state is an extended state that can only be accessed in ring 0. XSAVE feature set has been enhanced to manage supervisor states. Two new ring 0 instructions, XSAVES/XRSTORS, are introduced to save/restore supervisor states along with other XSAVE managed states. They are privileged instruction and only operate in compacted format. System programming for managing supervisor states in described in section 15.7.

Each XSAVE managed features may have additional feature specific system programming requirements such as exception handlers etc. Feature specific system programming requirements for XSAVE managed features are described in Section 15.8.

15.1 PROVIDING OPERATING SYSTEM SUPPORT FOR SSE EXTENSIONS

To use SSE extensions, the operating system or executive must provide support for initializing the processor to use these extensions, for handling SIMD floating-point exceptions, and for using FXSAVE and FXRSTOR (Section 10.5 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1) to manage context. XSAVE feature set can also be used to manage SSE state along with other processor extended states as described in 15.5. This section primarily focuses on using FXSAVE/FXRSTOR to manage SSE state. Because SSE extensions share the same state, experience the same sets of non-numerical and numerical exception behavior, these guidelines that

1. The collection also includes PCLMULQDQ and AES instructions operating on XMM state.

apply to SSE also apply to other sets of SIMD extensions that operate on the same processor state and subject to the same sets of non-numerical and numerical exception behavior.

Chapter 11, "Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2)," and Chapter 12, "Programming with Intel® SSE3, SSSE3, Intel® SSE4, and Intel® AES-NI," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, provide details on the Intel SSE instruction set.

15.1.1 Adding Support to an Operating System for SSE Extensions

The following guidelines describe functions that an operating system or executive must perform to support SSE extensions:

1. Check that the processor supports the SSE extensions.
2. Check that the processor supports the FXSAVE and FXRSTOR instructions or the XSAVE feature set.
3. Provide an initialization for the SSE states.
4. Provide support for the FXSAVE and FXRSTOR instructions or the XSAVE feature set.
5. Provide support (if necessary) in non-numeric exception handlers for exceptions generated by the SSE instructions.
6. Provide an exception handler for the SIMD floating-point exception (#XM).

The following sections describe how to implement each of these guidelines.

15.1.2 Checking for CPU Support

If the processor attempts to execute an unsupported SSE instruction, the processor generates an invalid-opcode exception (#UD). Before an operating system or executive attempts to use SSE extensions, it should check that support is present by confirming the following bit values returned by the CPUID instruction:

- CPUID.1:EDX.SSE[bit 25] = 1
- CPUID.1:EDX.SSE2[bit 26] = 1
- CPUID.1:ECX.SSE3[bit 0] = 1
- CPUID.1:ECX.SSSE3[bit 9] = 1
- CPUID.1:ECX.SSE4_1[bit 19] = 1
- CPUID.1:ECX.SSE4_2[bit 20] = 1

(To use POPCNT instruction, software must check CPUID.1:ECX.POPCNT[bit 23] = 1.)

Separate checks must be made to ensure that the processor supports either FXSAVE and FXRSTOR or the XSAVE feature set. See Section 10.5 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, and Chapter 13 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, respectively.

15.1.3 Initialization of the SSE Extensions

The operating system or executive should carry out the following steps to set up SSE extensions for use by application programs:

1. Set CR4.OSFXSR[bit 9] = 1. Setting this flag implies that the operating system provides facilities for saving and restoring SSE state using FXSAVE and FXRSTOR instructions. These instructions may be used to save the SSE state during task switches and when invoking the SIMD floating-point exception (#XM) handler (see Section 15.1.5, "Providing a Handler for the SIMD Floating-Point Exception (#XM)").

   If the processor does not support the FXSAVE and FXRSTOR instructions, attempting to set the OSFXSR flag causes a general-protection exception (#GP) to be generated.

2. Set CR4.OSXMMEXCPT[bit 10] = 1. Setting this flag implies that the operating system provides a SIMD floating-point exception (#XM) handler (see Section 15.1.5, "Providing a Handler for the SIMD Floating-Point Exception (#XM)").

NOTE

The OSFXSR and OSXMMEXCPT bits in control register CR4 must be set by the operating system. The processor has no other way of detecting operating-system support for the FXSAVE and FXRSTOR instructions or for handling SIMD floating-point exceptions.

3. Clear CR0.EM[bit 2] = 0. This action disables emulation of the x87 FPU, which is required when executing SSE instructions (see Section 2.5, "Control Registers").
4. Set CR0.MP[bit 1] = 1. This setting is required for Intel 64 and IA-32 processors that support the SSE extensions (see Section 11.2.1, "Configuring the x87 FPU Environment").

Table 15-1 and Table 15-2 show the actions of the processor when an SSE instruction is executed, depending on the following:

- OSFXSR and OSXMMEXCPT flags in control register CR4
- SSE/SSE2/SSE3/SSSE3/SSE4 feature flags returned by CPUID
- EM, MP, and TS flags in control register CR0

Table 15-1. Action Taken for Combinations of OSFXSR, OSXMMEXCPT, SSE, SSE2, SSE3, EM, MP, and TS[1]

CR4		CPUID	CR0 Flags			Action
OSFXSR	OSXMMEXCPT	SSE, SSE2, SSE3[2], SSE4_1[3]	EM	MP[4]	TS	
0	X[5]	X	X	1	X	#UD exception.
1	X	0	X	1	X	#UD exception.
1	X	1	1	1	X	#UD exception.
1	0	1	0	1	0	Execute instruction; #UD exception if unmasked SIMD floating-point exception is detected.
1	1	1	0	1	0	Execute instruction; #XM exception if unmasked SIMD floating-point exception is detected.
1	X	1	0	1	1	#NM exception.

NOTES:

1. For execution of any SSE instruction except the PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, MOVNTI, and CLFLUSH instructions.
2. Exception conditions due to CR4.OSFXSR or CR4.OSXMMEXCPT do not apply to FISTTP.
3. Only applies to DPPS, DPPD, ROUNDPS, ROUNDPD, ROUNDSS, ROUNDSD.
4. For processors that support the MMX instructions, the MP flag should be set.
5. X = Don't care.

Table 15-2. Action Taken for Combinations of OSFXSR, SSSE3, SSE4, EM, and TS

CR4	CPUID	CR0 Flags		Action
OSFXSR	SSSE3 SSE4_1[1] SSE4_2[2]	EM	TS	
0	X[3]	X	X	#UD exception.
1	0	X	X	#UD exception.
1	1	1	X	#UD exception.
1	1	0	1	#NM exception.

NOTES:

1. Applies to SSE4_1 instructions except DPPS, DPPD, ROUNDPS, ROUNDPD, ROUNDSS, ROUNDSD.
2. Applies to SSE4_2 instructions except CRC32 and POPCNT.
3. X = Don't care.

The SIMD floating-point exception mask bits (bits 7 through 12), the flush-to-zero flag (bit 15), the denormals-are-zero flag (bit 6), and the rounding control field (bits 13 and 14) in the MXCSR register should be left in their default values of 0. This permits the application to determine how these features are to be used.

15.1.4 Providing Non-Numeric Exception Handlers for Exceptions Generated by the SSE Instructions

SSE instructions can generate the same type of memory-access exceptions (such as page faults and limit violations) and other non-numeric exceptions as other Intel 64 and IA-32 architecture instructions generate.

Ordinarily, existing exception handlers can handle these and other non-numeric exceptions without code modification. However, depending on the mechanisms used in existing exception handlers, some modifications might need to be made.

The SSE extensions can generate the non-numeric exceptions listed below:

- Memory Access Exceptions:
  - — Stack-segment fault (#SS).
  - — General protection exception (#GP). Executing most SSE instructions with an unaligned 128-bit memory reference generates a general-protection exception. (The MOVUPS and MOVUPD instructions allow unaligned a loads or stores of 128-bit memory locations, without generating a general-protection exception.) A 128-bit reference within the stack segment that is not aligned to a 16-byte boundary will also generate a general-protection exception, instead a stack-segment fault exception (#SS).
  - — Page fault (#PF).
  - — Alignment check (#AC). When enabled, this type of alignment check operates on operands that are less than 128-bits in size: 16-bit, 32-bit, and 64-bit. To enable the generation of alignment check exceptions, do the following:
    - Set the AM flag (bit 18 of control register CR0)
    - Set the AC flag (bit 18 of the EFLAGS register)
    - CPL must be 3

    If alignment check exceptions are enabled, 16-bit, 32-bit, and 64-bit misalignment will be detected for the MOVUPD and MOVUPS instructions; detection of 128-bit misalignment is not guaranteed and may vary with implementation.

- System Exceptions:
  - Invalid-opcode exception (#UD). This exception is generated when executing SSE instructions under the following conditions:
    - SSE/SSE2/SSE3/SSSE3/SSE4_1/SSE4_2 feature flags returned by CPUID are set to 0. This condition does not affect the CLFLUSH instruction, nor POPCNT.
    - The CLFSH feature flag returned by the CPUID instruction is set to 0. This exception condition only pertains to the execution of the CLFLUSH instruction.
    - The POPCNT feature flag returned by the CPUID instruction is set to 0. This exception condition only pertains to the execution of the POPCNT instruction.
    - The EM flag (bit 2) in control register CR0 is set to 1, regardless of the value of TS flag (bit 3) of CR0. This condition does not affect the PAUSE, PREFETCH*h*, MOVNTI, SFENCE, LFENCE, MFENCE, CLFLUSH, CRC32, and POPCNT instructions.
    - The OSFXSR flag (bit 9) in control register CR4 is set to 0. This condition does not affect the PSHUFW, MOVNTQ, MOVNTI, PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, CLFLUSH, CRC32, and POPCNT instructions.
    - Executing an instruction that causes a SIMD floating-point exception when the OSXMMEXCPT flag (bit 10) in control register CR4 is set to 0. See Section 15.4.1, "Using the TS Flag to Control the Saving of the x87 FPU and SSE State."
  - Device not available (#NM). This exception is generated by executing a SSE instruction when the TS flag (bit 3) of CR0 is set to 1.

Other exceptions can occur during delivery of the above exceptions.

15.1.5 Providing a Handler for the SIMD Floating-Point Exception (#XM)

SSE instructions do not generate numeric exceptions on packed integer operations. They can generate the following numeric (SIMD floating-point) exceptions on packed and scalar single precision and double precision floating-point operations.

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

These SIMD floating-point exceptions (with the exception of the denormal operand exception) are defined in the IEEE Standard 754 for Floating-Point Arithmetic and represent the same conditions that cause x87 FPU floating-point error exceptions (#MF) to be generated for x87 FPU instructions.

Each of these exceptions can be masked, in which case the processor returns a reasonable result to the destination operand without invoking an exception handler. However, if any of these exceptions are left unmasked, detection of the exception condition results in a SIMD floating-point exception (#XM) being generated. See Chapter 7, "Interrupt 19—SIMD Floating-Point Exception (#XM)."

To handle unmasked SIMD floating-point exceptions, the operating system or executive must provide an exception handler. The section titled "SSE and SSE2 SIMD Floating-Point Exceptions" in Chapter 11, "Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2)," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, describes the SIMD floating-point exception classes and gives suggestions for writing an exception handler to handle them.

To indicate that the operating system provides a handler for SIMD floating-point exceptions (#XM), the OSXMMEXCPT flag (bit 10) must be set in control register CR4.

15.1.5.1 Numeric Error flag and IGNNE#

SSE extensions ignore the NE flag in control register CR0 (that is, they treat it as if it were always set) and the IGNNE# pin. When an unmasked SIMD floating-point exception is detected, it is always reported by generating a SIMD floating-point exception (#XM).

15.2 EMULATION OF SSE EXTENSIONS

The Intel 64 and IA-32 architectures do not support emulation of the SSE instructions, as they do for x87 FPU instructions.

The EM flag in control register CR0 (provided to invoke emulation of x87 FPU instructions) cannot be used to invoke emulation of SSE instructions. If an SSE instruction is executed when CR0.EM = 1, an invalid opcode exception (#UD) is generated. See Table 15-1.

15.3 SAVING AND RESTORING SSE STATE

The SSE state consists of the state of the XMM and MXCSR registers. Intel recommends the following method for saving and restoring this state:

- Execute the FXSAVE instruction to save the state of the XMM and MXCSR registers to memory.
- Execute the FXRSTOR instruction to restore the state of the XMM and MXCSR registers from the image saved in memory earlier.

This save and restore method is required for all operating systems. XSAVE feature set can also be used to save/restore SSE state. See Section 15.5, "The XSAVE Feature Set and Processor Extended State Management," for using the XSAVE feature set to save/restore SSE state.

In some cases, applications may choose to save only the XMM and MXCSR registers in the following manner:

- Execute MOVDQ instructions to save the contents of the XMM registers to memory.
- Execute a STMXCSR instruction to save the state of the MXCSR register to memory.

Such applications must restore the XMM and MXCSR registers as follows:

- Execute MOVDQ instructions to load the saved contents of the XMM registers from memory into the XMM registers.
- Execute a LDMXCSR instruction to restore the state of the MXCSR register from memory.

15.4 DESIGNING OS FACILITIES FOR SAVING X87 FPU, SSE, AND EXTENDED STATES ON TASK OR CONTEXT SWITCHES

The x87 FPU and SSE state consist of the state of the x87 FPU, XMM, and MXCSR registers. The FXSAVE and FXRSTOR instructions provide a fast method for saving and restoring this state. The XSAVE feature set can also be used to save FP and SSE state along with other extended states (see Section 15.5).

Older operating systems may use FSAVE/FNSAVE and FRSTOR to save the x87 FPU state. These facilities can be extended to save and restore SSE state by substituting FXSAVE and FXRSTOR or the XSAVE feature set in place of FSAVE/FNSAVE and FRSTOR.

If task or context switching facilities are written from scratch, any of several approaches may be taken for using the FXSAVE and FXRSTOR instructions or the XSAVE feature set to save and restore x87 FPU and SSE state:

- The operating system can require applications that are intended to be run as tasks take responsibility for saving the states prior to a task suspension during a task switch and for restoring the states when the task is resumed. This approach is appropriate for cooperative multitasking operating systems, where the application has control over (or is able to determine) when a task switch is about to occur and can save state prior to the task switch.

- The operating system can take the responsibility for saving the states as part of the task switch process and restoring the state of the registers when a suspended task is resumed. This approach is appropriate for preemptive multitasking operating systems, where the application cannot know when it is going to be preempted and cannot prepare in advance for task switching.
- The operating system can take the responsibility for saving the states as part of the task switch process, but delay the restoring of the states until an instruction operating on the states is actually executed by the new task. See Section 15.4.1, "Using the TS Flag to Control the Saving of the x87 FPU and SSE State," for more information. This approach is called lazy restore.

  The use of lazy restore mechanism in context switches is not recommended when XSAVE feature set is used to save/restore states for the following reasons.

  — With XSAVE feature set, Intel processors have optimizations in place to avoid saving the state components that are in their initial configurations or when they have not been modified since they were restored last. These optimizations eliminate the need for lazy restore. See section 13.5.4 in Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

  — Intel processors have power optimizations when state components are in their initial configurations. Use of lazy restore retains the non-initial configuration of the last thread and is not power efficient.

  — Not all extended states support lazy restore mechanisms. As such, when one or more such states are enabled it becomes very inefficient to use lazy restore as it results in two separate state restore, one in context switch for the states that does not support lazy restore and one in the #NM handler for states that support lazy restore.

15.4.1 Using the TS Flag to Control the Saving of the x87 FPU and SSE State

The TS flag in control register CR0 is provided to allow the operating system to delay saving/restoring the x87 FPU and SSE state until an instruction that actually accesses this state is encountered in a new task. When the TS flag is set, the processor monitors the instruction stream for x87 FPU, MMX, SSE instructions. When the processor detects one of these instructions, it raises a device-not-available exception (#NM) prior to executing the instruction. The #NM exception handler can then be used to save the x87 FPU and SSE state for the previous task (using an FXSAVE, XSAVE, or XSAVEOPT instruction) and load the x87 FPU and SSE state for the current task (using an FXRSTOR or XRSOTR instruction). If the task never encounters an x87 FPU, MMX, or SSE instruction, the device-not-available exception will not be raised and a task state will not be saved/restored unnecessarily.

NOTE

The CRC32 and POPCNT instructions do not operate on the x87 FPU or SSE state. They operate on the general-purpose registers and are not involved with the techniques described above.

The TS flag can be set either explicitly (by executing a MOV instruction to control register CR0) or implicitly (using the IA-32 architecture's native task switching mechanism). When the native task switching mechanism is used, the processor automatically sets the TS flag on a task switch. After the device-not-available handler has saved the x87 FPU and SSE state, it should execute the CLTS instruction to clear the TS flag.

15.5 THE XSAVE FEATURE SET AND PROCESSOR EXTENDED STATE MANAGEMENT

The architecture of XSAVE feature set is described in Chapter 13 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. The XSAVE feature set includes the following:

- An extensible data layout for existing and future processor state extensions. The layout of the XSAVE area extends from the 512-byte FXSAVE/FXRSTOR layout to provide compatibility and migration path from managing the legacy FXSAVE/FXRSTOR area. The XSAVE area is described in more detail in Section 13.4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.
- CPUID enhancements for feature enumeration. See Section 13.2 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

- Control register enhancement and dedicated register for enabling each processor extended state. See Section 13.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.
- Instructions to save state to and restore state from the XSAVE area. See Section 13.7 through Section 13.9 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

Operating systems can utilize XSAVE feature set to manage both FP/SSE state and processor extended states. CPUID leaf 0DH enumerates XSAVE feature set related information. The following guidelines provide the steps an operating system needs to take to support legacy FP/SSE states and processor extended states.

1. Check that the processor supports the XSAVE feature set
2. Determine the set of XSAVE managed features that the operating system intends to enable and calculate the size of the buffer needed to save/restore the states during context switch and other flows
3. Enable use of XSAVE feature set and XSAVE managed features
4. Provide an initialization for the XSAVE managed feature state components
5. Provide (if necessary) required exception handlers for exceptions generated each of the XSAVE managed features.

15.5.1 Checking the Support for XSAVE Feature Set

Support for XSAVE Feature set is enumerated in CPUID.1.ECX.XSAVE[bit 26]. Enumeration of this bit indicates that the processor supports XSAVE/XRSTOR instructions to manage state and XSETBV/XGETBV on XCR0 to enable and get enabled states. An operating system needs to enable XSAVE feature set as described later.

Additionally, CPUID.(EAX=0DH, ECX=1).EAX enumerates additional XSAVE sub features such as optimized save, compaction, and supervisor state support. The following table summarizes XSAVE sub features. Once an operating system enables XSAVE feature set, all the sub-features enumerated are also available. There is no need to enable each additional sub feature.

Table 15-3. CPUID.(EAX=0DH, ECX=1) EAX Bit Assignment

EAX Bit Position	Meaning
0	If set, indicates availability of the XSAVEOPT instruction.
1	If set, indicates availability of the XSAVEC instruction and the corresponding compaction enhancements to the legacy XRSTOR instruction.
2	If set, indicates support for execution of XGETBV with ECX=1. This execution returns the state-component bitmap XINUSE. If XINUSE[i] = 0, state component i is in its initial configuration. Execution of XSETBV with ECX=1 causes a #GP.
3	If set, indicates support for XSAVES/XRSTORS and IA32_XSS MSR
31:4	Reserved

15.5.2 Determining the XSAVE Managed Feature States And The Required Buffer Size

Each XSAVE managed feature has one or more state components associated with it. An operating system policy needs to determine the XSAVE managed features to support and determine the corresponding state components to enable. When determining the XSAVE managed features to support, operating system needs to take into account the dependencies between them (e.g., AVX feature depends on SSE feature). Similarly, when a XSAVE managed feature has more than one state component, all of them need to be enabled. Each logical processor enumerates supported XSAVE state components in CPUID.(EAX=0DH, ECX=0).EDX:EAX. An operating system may enable all or a subset of the state components enumerated by the processor based on the OS policy.

The size of the memory buffer needed to save enabled XSAVE state components depends on whether the OS opts-in to use compacted format or not. Section 13.4.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, describes the layout of the extended region of the XSAVE area.

15.5.3 Enable the Use Of XSAVE Feature Set And XSAVE State Components

Operating systems need to enable the use of XSAVE feature set by writing to CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR. When the XSAVE feature set is enabled, all enumerated XSAVE sub features such as optimized save, compaction, and supervisor state support are also enabled. Operating systems also need to enable the XSAVE state components in XCR0 using the XSETBV instruction.

XSAVE state components can subsequently be disabled in XCR0. However, disabling state components of AVX or AVX-512 that are not in initial configuration may incur power and performance penalty on SSE and AVX instructions respectively. If AVX state is disabled when it is not in its initial configuration, subsequent SSE instructions may incur a penalty. If AVX-512 state is disabled when it is not in its initial configuration, subsequent SSE and AVX instructions may incur a penalty. It is recommended that the operating systems and VMM set AVX or AVX-512 state components to their initial configuration before disabling them. This can be achieved by one of the two methods below.

- Using XRSTOR: Operating system or VMM can set the state of AVX or AVX-512 state components using XRSTOR instruction before disabling them in XCR0.
- Using VZEROUPPER: Operating system or VMM can set AVX and AVX-512 state components to their initial configuration using VZEROUPPER instruction before disabling them in XCR0. Note that this will set both AVX and AVX-512 state components to their initial configuration. If the intent is to only disable AVX-512 state, Operating system or VMM will need to save AVX state before executing VZEROUPPER and restore it afterwards.

15.5.4 Provide an Initialization for the XSAVE State Components

The XSAVE header of a newly allocated XSAVE area should be initialized to all zeroes before saving context. An operating system may choose to establish beginning state-component values for a task by executing XRSTOR from an XSAVE area that the OS has configured. If it is desired to begin state component i in its initial configuration, the OS should clear bit i in the XSTATE_BV field in the XSAVE header; otherwise, it should set that bit and place the desired beginning value in the appropriate location in the XSAVE area.

When a buffer is allocated for compacted size, software must ensure that the XCOMP_BV field is setup correctly before restoring from the buffer. Bit 63 of the XCOMP_BV field indicates that the save area is in the compacted format and the remaining bits indicate the states that have space allocated in the save area. If the buffer is first used to save the state in compacted format, then the save instructions will setup the XCOMP_BV field appropriately. If the buffer is first used to restore the state, then software must set up the XCOMP_BV field.

15.5.5 Providing the Required Exception Handlers

Instructions part of each XSAVE managed features may generate exceptions and operating system may need to enable such exceptions and provide handlers for them. Section 15.8 describes feature specific OS requirements for each XSAVE managed features.

15.6 INTEROPERABILITY OF THE XSAVE FEATURE SET AND FXSAVE/FXRSTOR

The FXSAVE instruction writes x87 FPU and SSE state information to a 512-byte FXSAVE save area. FXRSTOR restores the processor's x87 FPU and SSE states from an FXSAVE area. The XSAVE features set supports x87 FPU and SSE states using the same layout as the FXSAVE area to provide interoperability of FXSAVE versus XSAVE, and FXRSTOR versus XRSTOR. The XSAVE feature set allows system software to manage SSE state independent of x87 FPU states. Thus system software that had been using FXSAVE and FXRSTOR to manage x87 FPU and SSE states can transition to using the XSAVE feature set to manage x87 FPU, SSE, and other processor extended states in a systematic and forward-looking manner. See Section 10.5 and Chapter 13 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more details.

System software can implement forward-looking processor extended state management using the XSAVE feature set. In this case, system software must specify the bit vector mask in EDX:EAX appropriately when executing XSAVE/XRSTOR instructions.

For instance, the OS can supply instructions in the XSAVE feature set with a bit vector in EDX:EAX with the two least significant bits (corresponding to x87 FPU and SSE state) equal to 0. Then, the XSAVE instruction will not write the processor's x87 FPU and SSE state into memory. Similarly, the XRSTOR instruction executed with a value in EDX:EAX with the least two significant bit equal to 0 will not restore nor initialize the processor's x87 FPU and SSE state.

The processor's action as a result of executing XRSTOR is given in Section 13.8 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1. The instruction may be used to initialize x87 FPU or XMM registers. When the MXCSR register is updated from memory, reserved bit checking is enforced. The saving/restoring of MXCSR is bound to the SSE state, independent of the x87 FPU state. The action of XSAVE is given in Section 13.7 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

15.7 THE XSAVE FEATURE SET AND PROCESSOR SUPERVISOR STATE MANAGEMENT

Supervisor state is a processor state that is only accessible in ring 0. An extension to the XSAVE feature set, enumerated by CPUID.(EAX=0DH, ECX=1).EAX[bit 3] allows the management of the supervisor states using the XSAVE feature set. See Chapter 13 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for the details of the supervisor state XSAVE feature set extension. The supervisor state extension includes the following:

- CPUID enhancements to enumerate the set of supervisor states and their sizes that can be managed by the XSAVE feature set.
- The IA32_XSS MSR to enable the XSAVE feature set to manage one or more enumerated supervisor states.
- A pair of privileged save/restore instructions, XSAVES and XRSTORS, to save/restore supervisor states along with other XSAVE managed feature states.

The guidelines to enable the XSAVE feature set to manage supervisor state are very similar to the steps outlined in Section 13.6 with the differences noted below. The set of supervisor states that can be managed by the XSAVE feature set is enumerated in (EAX=0DH, ECX=1).EDX:ECX. XSAVE managed supervisor states are enabled in the IA32_XSS MSR instead of the XCR0 control register. There are semantic differences between user states enabled in XCR0 and supervisor state enabled in the IA32_XSS MSR. A supervisor state enabled in the IA32_XSS MSR:

- May be accessed via other mechanisms such as RDMSR/WRMSR even when they are not enabled in the IA32_XSS MSR. Enabling a supervisor state in the IA32_XSS MSR merely indicates that the state can be saved/restored using XSAVES/XRSTORS instructions.
- May have side effects when saving/restoring the state such as disabling/enabling the feature associated with the state. This behavior is feature specific and will be documented along with the feature description.
- May generate faults when saving/restoring the state. XSAVES/XRSTORS will follow the faulting behavior of RDMSR/WRMSR respectively if the corresponding state is also accessible using RDMSR/WRMSR.
- XRSTORS may fault when restoring the state for supervisor features that are already enabled via feature specific mechanisms. This behavior is feature specific and will be documented along with the feature description.

When a supervisor state is disabled via a feature specific mechanism, the state does not automatically get marked as INIT. Hence XSAVES/XRSTORS will continue to save/restore the state subject to available optimizations. If the software does not intend to preserve the state when it disables the feature, it should initialize it to hardware INIT value with the XRSTORS instruction so that XSAVES/XRSTORS perform optimally for that state.

15.8 SYSTEM PROGRAMMING FOR XSAVE MANAGED FEATURES

This section describes system programming requirement for each XSAVE managed features that are feature specific, such as exception handling.

15.8.1 Intel® Advanced Vector Extensions (Intel® AVX)

Intel AVX instructions comprises of 256-bit and 128-bit instructions that operates on 256-bit YMM registers. The XSAVE feature set allows software to save and restore the state of these registers. See Chapter 13 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

For processors that support YMM states, the YMM state exists in all operating modes. However, the available instruction interfaces to access YMM states may vary in different modes.

Operating systems must use the XSAVE feature set for YMM state management. The XSAVE feature set also provides flexible and efficient interface to manage XMM/MXCSR states and x87 FPU states in conjunction with newer processor extended states like YMM states. Operating systems may need to be aware of the following when supporting AVX.

- Saving/Restoring AVX state in non-compacted format without SSE state will also save/restore MXCSR even though MXCSR is not part of AVX state. This does not happen when compacted format is used.
- Few AVX instructions such as VZEROUPPER/VZEROALL may operate on future expansion of YMM registers.

An operating system must enable its YMM state management to support AVX and any 256-bit extensions that operate on YMM registers. Otherwise, an attempt to execute an instruction in AVX extensions (including an enhanced 128-bit SIMD instructions using VEX encoding) will cause a #UD exception.

AVX instructions may generate SIMD floating-point exceptions. An OS must enable SIMD floating-point exception support by setting CR4.OSXMMEXCPT[bit 10]=1.

15.8.2 Intel® Advanced Vector Extensions 512 (Intel® AVX-512)

Intel AVX-512 instructions are encoded using EVEX prefix. The EVEX encoding scheme can support 512-bit, 256-bit and 128-bit instructions that operate on opmask, ZMM, YMM, and XMM registers.

For processors that support the Intel AVX-512 family of instructions, the extended processor states (ZMM and opmask registers) exist in all operating modes. However, the access to these states may vary in different modes. The processor's support for instruction extensions that employ EVEX prefix encoding is independent of the processor's support for using XSAVE feature set on those states.

Instructions requiring EVEX prefix encoding are generally supported in 64-bit, 32-bit modes, and 16-bit protected mode. They are not supported in Real mode, Virtual-8086 mode or entering into SMM mode. Note that bits MAXVL-1:256 (511:256) of ZMM register state are maintained across transitions into and out of these modes. Because the XSAVE feature set instruction can operate in all operating modes, it is possible that the processor's ZMM register state can be modified by software in any operating mode by executing XRSTOR.

Operating systems must use the XSAVE/XRSTOR/XSAVEOPT instructions for ZMM and opmask state management. An OS must enable its ZMM and opmask state management to support Intel AVX-512 Foundation instructions. Otherwise, an attempt to execute an instruction in Intel AVX-512 Foundation instructions (including a scalar 128-bit SIMD instructions using EVEX encoding) will cause a #UD exception. An operating system, which enables the AVX-512 state to support Intel AVX-512 Foundation instructions, is also sufficient to support the rest of the Intel AVX-512 family of instructions. Note that even though ZMM8-ZMM31 are not accessible in 32 bit mode, a 32 bit OS is still required to allocate memory for the entire ZMM state.

Intel AVX-512 Foundation instructions may generate SIMD floating-point exceptions. An OS must enable SIMD floating-point exception support by setting CR4.OSXMMEXCPT[bit 10]=1.