MindShare Technology Series

x86 Instruction Set Architecture

Comprehensive 32- and 64-bit Coverage

Tom Shanley | MindShare, Inc.

x86 Instruction Set Architecture

Comprehensive 32/64-bit Coverage

First Edition

Also by Tom Shanley

HEAVEN'S FAVORITE
—A Novel of Genghis Khan—
Book 1, **ASCENT: THE RISE OF CHINGGIS KHAN**
Book 2, **DOMINION: DAWN OF THE MONGOL EMPIRE**

MINDSHARE TECHNICAL TRAINING

Please visit www.mindshare.com for a complete description of Mind-Share's technical offerings:

- Books
- eBooks
- eLearning modules
- Public courses
- On-site course
- On-line courses

Intel Core 2 Processor (Penryn)
Intel Nehalem Processor
Intel Atom Processor
AMD Opteron Processor (Barcelona)
Intel 32/64-bit x86 Software Architecture
AMD 32/64-bit x86 Software Architecture
x86 Assembly Language Programming
Protected Mode Programming
PC Virtualization
IO Virtualization (IOV)
Computer Architectures with Intel Chipsets
Intel QuickPath Interconnect (QPI)
PCI Express 2.0
USB 2.0
USB 3.0
Embedded USB 2.0 Workshop
PCI
PCI-X
Modern DRAM Architecture
SAS
Serial ATA
High Speed Design
EMI / EMC
Bluetooth Wireless Product Development
SMT Manufacturing
SMT Testing

x86 Instruction Set Architecture

Comprehensive 32/64-bit Coverage

First Edition

MINDSHARE, INC.

TOM SHANLEY

MindShare Press

Colorado Springs, USA

Refer to “Trademarks” on page 5 for trademark information.

ISBN: 0-9770878-5-3

Cover Design: Michelle Petrie
Set in 10 point Palatino by MindShare, Inc.

First Printing, December 2009

MindShare Press books are available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Special Sales Department at (575)-373-0336.

Find MindShare Press on the World Wide Web at:
http://www.mindshare.com/

To Nancy, the strongest person I know.

With Love,

Tom

P. S. It’s done. I’m back.

At-a-Glance Table of Contents

Part 1: Introduction, intended as a back-drop to the detailed discussions that follow, consists of the following chapters:

Part 2: IA-32 Mode provides a detailed description of two IA-32 Mode sub-modes—Real Mode and Protected Mode—and consists of the following chapters:

Part 3: IA-32e OS Kernel Environment provides a detailed description of the IA-32e OS kernel environment and consists of the following chapters:

- Chapter 23, "IA-32e OS Environment," on page 913.
- Chapter 24, "IA-32e Address Translation," on page 983.

Part 4: Compatibility Mode provides a detailed description of the Compatibility submode of IA-32e Mode and consist of the following chapter:

- Chapter 25, "Compatibility Mode," on page 1009.

Part 5: 64-bit Mode provides a detailed description of the 64-bit submode of IA-32e Mode and consists of the following chapters:

- Chapter 26, "64-bit Register Overview," on page 1023.
- Chapter 27, "64-bit Operands and Addressing," on page 1041.
- Chapter 28, "64-bit Odds and Ends," on page 1075.

Part 6: Mode Switching Detail provides a detailed description of:

- Switching from Real Mode to Protected Mode. This topic is covered in Chapter 29, "Transitioning to Protected Mode," on page 1113.
- Switching from Protected Mode to IA-32e Mode. This topic is covered in Chapter 30, "Transitioning to IA-32e Mode," on page 1139.

Part 7: Other Topics provides detailed descriptions of the following topics:

- Chapter 31, "Introduction to Virtualization Technology," on page 1147.
- Chapter 32, "System Management Mode (SMM)," on page 1167.
- Chapter 33, "Machine Check Architecture (MCA)," on page 1207.
- Chapter 34, "The Local and IO APICs," on page 1239.

About This Book

Part 1: Introduction

Chapter 1: Basic Terms and Concepts

Chapter 2: Mode/SubMode Introduction

Contents

Chapter 3: A (very) Brief History

Chapter 4: State After Reset

Part 2: IA-32 Mode

Chapter 5: Intro to the IA-32 Ecosystem

Chapter 6: Instruction Set Expansion

Contents

Chapter 7: 32-bit Machine Language Instruction Format

Chapter 8: Real Mode (8086 Emulation)

Chapter 9: Legacy x87 FP Support

Chapter 10: Introduction to Multitasking

Chapter 11: Multitasking-Related Issues

Chapter 14: Code, Calls and Privilege Checks

Chapter 15: Data and Stack Segments

Chapter 16: IA-32 Address Translation Mechanisms

Chapter 17: Memory Type Configuration

Chapter 18: Task Switching

Chapter 19: Protected Mode Interrupts and Exceptions

Contents

Chapter 20: Virtual 8086 Mode

Chapter 21: The MMX Facilities

Chapter 22: The SSE Facilities

Chapter 24: IA-32e Address Translation

Part 4: Compatibility Mode

Chapter 25: Compatibility Mode

Part 5: 64-bit Mode

Chapter 26: 64-bit Register Overview

Chapter 27: 64-bit Operands and Addressing

Chapter 28: 64-bit Odds and Ends

Part 6: Mode Switching Detail

Chapter 29: Transitioning to Protected Mode

Contents

Chapter 32: System Management Mode (SMM)

Chapter 33: Machine Check Architecture (MCA)

Chapter 34: The Local and IO APICs

Contents

Contents

Tables

Tables

About This Book

Is This the Book for You?

If you're looking for a book designed specifically for those who need to come up to speed on the 32-/64-bit x86 Instruction Set Architecture (ISA) as quickly and painlessly as possible, then consider this book.

On the other hand, the author fully realizes that a certain segment of the technical population rejects and is, indeed, deeply offended by any attempt to present arcane technical material in a learning-friendly manner. Having been exposed to the occasional criticism from such individuals, if you fall into this category I can only forewarn that this book is not for you. Do not waste your money or your time.

A Moving Target

The reader should keep in mind that MindShare books often deal with rapidly evolving technologies. This being the case, it should be recognized that this book is a *snapshot* of the state of the x86 programming environment at the time that the book was completed (November, 2009).

x86 Instruction Set Architecture (ISA)

Throughout this book, the term *ISA* (*Instruction Set Architecture)* refers to the current execution environment defined by the x86 ISA specification:

- Any reference to the term *IA-32 ISA* refers to the facilities visible to the programmer when the processor is operating in 32-bit mode (referred to by Intel as IA-32 Mode and by AMD as Legacy Mode).
- Any reference to the term *Intel 64 ISA* refers to the facilities visible to the programmer when the processor is operating in 64-bit Mode (referred to by Intel as Intel 64 Mode and by AMD as Long Mode).

Glossary of Terms

A comprehensive glossary may be found on page 1391.

32-/64-bit x86 Instruction Set Architecture Specification

As of this writing (February, 2009), the ISA specification is embodied in the *Intel 64 and IA-32 Architectures Software Developer's Manual* which currently consist of the following five volumes:

- *Basic Architecture*; order number 253665.
- *Instruction Set Reference A-M*; order number 253666.
- *Instruction Set Reference N-Z*; order number 253667.
- *System Programming Guide, Part 1*; order number 253668.
- *System Programming Guide, Part 2*; order number 253669.

Alternatively, the specification is also embodied in the equivalent manuals available from AMD.

While the specification does define the register and instruction sets, interrupt and software exception handling, and standard processor facilities such as memory address generation and translation, the processor modes of operation, multitasking and protection mechanisms, etc., it does *not* specify processor-specific features such as the following:

- Whether or not a processor design includes caches and, if so, the number of, size of, and architecture of the caches.
- Whether or not a processor design includes one or more TLBs (Translation Lookaside Buffers) and, if so, the number of, size of, and architecture of the TLBs.
- The type of interface that connects the processor to the system.
- The number and types of instruction execution units.
- The implementation-specific aspects of a processor's microarchitecture.
- Various other performance enhancement features (branch prediction mechanisms, etc.).

The Specification Is the Final Word

This book represents the author's interpretation of the Intel x86 ISA specification. When in doubt, the Intel specification is the final word.

Book Organization

This book is organized in seven parts:

“Part 1: Introduction”, intended as a back-drop to the detailed discussions that follow, consists of the following chapters:

“Part 2: IA-32 Mode” provides a detailed description of two IA-32 Mode sub-modes—Real Mode and Protected Mode—and consists of the following chapters:

“Part 3: IA-32e OS Kernel Environment” provides a detailed description of the IA-32e OS kernel environment and consists of the following chapters:

“Part 4: Compatibility Mode” provides a detailed description of the Compatibility submode of IA-32e Mode and consist of the following chapter:

- Chapter 25, entitled "Compatibility Mode," on page 1009.

“Part 5: 64-bit Mode” provides a detailed description of the 64-bit submode of IA-32e Mode and consists of the following chapters:

- Chapter 26, entitled "64-bit Register Overview," on page 1023.
- Chapter 27, entitled "64-bit Operands and Addressing," on page 1041.
- Chapter 28, entitled "64-bit Odds and Ends," on page 1075.

“Part 6: Mode Switching Detail” provides a detailed description of:

- Switching from Real Mode to Protected Mode. This topic is covered in Chapter 29, entitled "Transitioning to Protected Mode," on page 1113.
- Switching from Protected Mode to IA-32e Mode. This topic is covered in Chapter 30, entitled "Transitioning to IA-32e Mode," on page 1139.

“Part 7: Other Topics” provides detailed descriptions of the following topics:

- Chapter 31, entitled "Introduction to Virtualization Technology," on page 1147.
- Chapter 32, entitled "System Management Mode (SMM)," on page 1167.
- Chapter 33, entitled "Machine Check Architecture (MCA)," on page 1207.
- Chapter 34, entitled "The Local and IO APICs," on page 1239.

Topics Outside the Scope of This Book

The CPUID Instruction

The CPUID instruction is referred to numerous times in this book. For a detailed description of its usage, refer to the Intel publication entitled *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A, Instruction Set Reference A-M*.

Detailed Description of Hyper-Threading

The Intel Hyper-Threading facility is not covered in this book because it is not part of the x86 ISA. For a detailed description of this facility, refer to Chapter 39 in the MindShare book entitled *The Unabridged Pentium 4*.

Detailed Description of Performance Monitoring

The Intel Performance Monitoring facility is not covered in this book because it is not part of the x86 ISA. For a detailed description of this facility, refer to the section entitled *The Performance Monitoring Facility* in Chapter 56 of the MindShare book entitled *The Unabridged Pentium 4*.

Documentation Conventions

The conventions used in this book for numeric values are defined below:

- Hexadecimal Notation. All hex numbers are followed by an "h." Examples:
  - 9A4Eh
  - 0100h
- Binary Notation. All binary numbers are followed by a "b." Examples:
  - 0001 0101b
  - 01b
- Decimal Notation. Numbers without any suffix are decimal. When required for clarity, decimal numbers may be followed by a *d*. Examples:
  - 16
  - 255
  - 256d
  - 128d

Other commonly used designations are defined below:

- *lsb* refers to the least-significant bit.
- *LSB* refers to the least-significant byte.
- *msb* refers to the most-significant bit.
- *MSB* refers to the most-significant byte.
- Bit Fields. In many cases, bit fields are documented in the following manner: CR0[15:8] refers to Control Register 0 bits 8 - 15.
- Notations such as *CSDesc[BaseAddress]* are interpreted as the *Base Address field in the Code Segment Descriptor*.

Trademarks

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Those trademark designations known to MindShare Press are listed in Table 1-1 on page 6.

Table 1-1: Trademarks

Trademarked Terms	Trademark Owner
AMD, AMD64, Opteron	AMD
Atom, Core, Core 2, Core 2 Duo, Core 2 Quad, Core 2 Solo, Core i7, Core Solo, Hyper-Threading, Intel, Itanium, MMX, NetBurst, Pentium, QPI or QuickPath Interconnect, SpeedStep, SSE, VTune, Xeon.	Intel
Apple, OS X	Apple Computer
FrameMaker	Adobe Systems
IBM, PC-AT, PS/2	IBM
Linux	Linus Torvalds
PCI, PCI Express, PCIe, PCI-X	PCI SIG
SIMD	?
Unix	The Open Group, SCO, ? (I'll leave this one to the lawyers; your guess is as good as mine)
Word	Microsoft

Visit Our Web Site

Our web site (***www.mindshare.com***) provides detailed descriptions of all of our products including:

- Books and ebooks.
- On-site classes.
- Public, open-enrollment classes.
- Virtual, instructor-led classes.
- Self-paced eLearning modules.
- Technical papers and the MindShare newsletter.

We Want Your Feedback

MindShare values your comments, questions and suggestions. You can contact us via mail, phone, fax or email.

Phone: (719) 487-1417 and in the U.S. (800) 633-1440
Fax: (719) 487-1434
Email: support@mindshare.com

Mailing Address:

MindShare, Inc.
4285 Slash Pine Drive
Colorado Springs, CO 80908

1 Basic Terms and Concepts

This Chapter

This chapter provides a basic definition of the Instruction Set Architecture (ISA), differentiates between the IA-32 and Intel 64 processor architectures, and defines some other important terms and concepts.

The Next Chapter

The next chapter introduces the execution modes and submodes as well as mode switching basics.

ISA Definition

Wikipedia Definition: The Instruction Set Architecture, or ISA, is defined as that part of the processor architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external IO.

This Book Focuses on the Common Intel/AMD ISA

With the exception of some small deviations and differences in terminology, the Intel and AMD x86 processors share a common ISA. This book focuses on their shared attributes and does not cover those areas where the two companies have chosen widely divergent, non-x86 ISA-compliant, solutions.

For Simplicity, Intel Terminology Is Used Throughout

Rather than confusing matters by using both Intel and AMD terminology throughout the book, the author has chosen to use only Intel terminology.

Some Terms in This Chapter May Be New To the Reader

Someone new to the x86 software environment will almost certainly encounter some unfamiliar terms in this chapter. Don't let it disturb you. Every term and concept *will* be described in detail at the appropriate place in the book. The important things to take away from this chapter are the broader concepts.

Two x86 ISA Architectures

All Intel x86-family processors introduced since the advent of the 386 can be divided into two categories (see Table 1-1 on page 13):

- Those that cannot execute 64-bit code—defined by Intel as ***IA-32 Architecture*** processors,
- and those that can—defined by Intel as ***Intel 64 Architecture*** processors.

This distinction is an important one but is not always referred to correctly—even in the vendor's own documentation. As an example, in section 3.2.1 of the *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture* manual, it states:

> "A task or program running in 64-bit mode on an IA-32 processor can address linear address space of up to 2^{64} bytes (subject to the canonical addressing requirement described in Section 3.3.7.1) and physical address space of up to 2^{40} bytes."

There is no such thing as 64-bit Mode on an IA-32 processor. Consistent use of terms is critical to a clear understanding of any subject. For someone learning the fundamentals of the x86 programming environment, misleading statements such as this can lead to monumental confusion.

Table 1-1: x86 Software Architectures

Processor Family	Description
x86 Software Architectures All Intel x86-family processors introduced since the advent of the 386 can be divided into two categories:	
IA-32 Processor	Implements only the Intel IA-32 Architecture which supports the execution of 16- and 32-bit x86 code.
Intel 64 Processor	Implements the Intel 64 Architecture, a superset of the IA-32 Architecture: • When operating in IA-32 Mode, the processor supports the execution of 16- and 32-bit x86 code. • When operating in IA-32e Mode, the processor supports the execution of 16-, 32- and 64-bit x86 code.

Processors, Cores and Logical Processors

For many, many years, life was simple: a physical processor package contained a single *core*: i.e, the engine that fetched machine language instructions (i.e., a program) from memory, decoded them, dispatched them to the appropriate execution units and then committed their results to the core's register set. This required:

- A single register set.
- A single set of execution units.
- A set of facilities to handle things like:
  - — Virtual-to-physical memory address translation.
  - — Interrupts and exceptions.
  - — Protection.
  - — etc.

The advent of multi-core processors and Hyper-Threading (more in a moment) has inevitably led to a confusion of terminology. As an example, consider the case where a dual core processor contains two cores each of which represents a stand-alone fetch, decode, dispatch, execution engine. Each implements its own register set, instruction fetcher, decoders, dispatcher, and execution units. So, in this scenario, the term *processor* really refers to a package containing two cores, each of which represents a separate processing engine. In all likelihood, though, the two cores may share some resources (typically, one or more caches).

Refer to Figure 1-1 on page 15. To further muddy the waters, a core may implement Hyper-Threading capability, in which case, from a programmer's perspective, a single core would implement two or more independent execution engines (referred to as logical processors):

- Each of which implements its own register set and dedicated resources. This includes a dedicated Local APIC (see "APIC" on page 19) to handle interrupt and exception events for its associated logical processor.
- All of which, invisible to software, may share some resources.

As if that's not confusing enough, if the physical processor's Hyper-Threading capability is disabled, then the second logical processor in each core (referred to as the *secondary logical processor;* the first is referred to as the *primary logical processor*) is disabled and each core functions as a single logical processor.

To sum it up, a physical processor contains one or more cores and, if it implements Hyper-Threading *and* it has been enabled, each core appears to software as two or more separate processors (i.e., logical processors). During a given period of time, all of the logical processors could be executing separate program threads.

Fundamental Processing Engine: Logical Processor

Rather than sprinkling hundreds of references to processors, cores and logical processors throughout the remainder of the book, the fundamental processing engine will heretofore be referred to as a *logical processor* (unless, of course, I am specifically discussing the physical processor package or a core, rather than a logical processor).

2 Mode/SubMode Introduction

The Previous Chapter

The previous chapter provided a basic definition of the Instruction Set Architecture (ISA), differentiated between the IA-32 and Intel 64 processor architectures, and defined some other important terms and concepts.

This Chapter

This chapter introduces the execution modes and submodes and mode switching basics.

The Next Chapter

The next chapter introduces the evolution of the x86 ISA, as well as the basic operational characteristics of 8086 Real Mode, 286 Protected Mode, and 386 Protected Mode. It also introduces the Intel microarchitecture families including a product introduction timeline.

Basic Execution Modes

Figure 2-1 on page 22 illustrates the execution modes supported on processors based on the IA-32 architecture versus those based on the Intel 64 architecture. Table 2-1 on page 23 provides an elementary description of the two basic execution modes—IA-32 Mode and IA-32e Mode.

Figure 2-1: Execution Mode Diagram

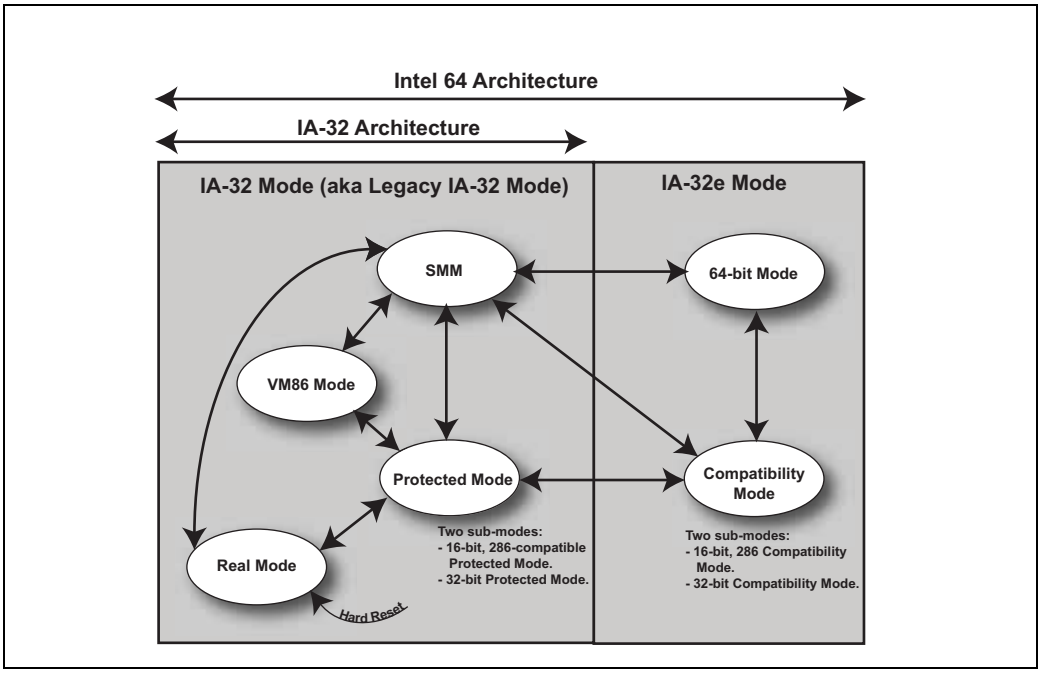


Table 2-1: Basic Execution Modes

Mode	Description
IA-32 Mode (also referred to as Legacy IA-32 Mode)	• IA-32 Architecture processors are always in IA-32 Mode which consists of the following SubModes: – Real Mode. – System Management Mode (SMM). – Protected Mode. – VM86 Mode. • At a given moment in time, an Intel 64 Architecture processor is operating in either: – IA-32 Mode, or – IA-32e (IA-32 Extended) Mode. "IA-32 SubModes" on page 25 describes the IA-32 execution SubModes. **Problems associated with IA-32 Mode**: Some of the problems associated with IA-32 Mode are: • The instruction set syntax uses a 3-bit field to specify a source or destination register. As a result, there are only eight addressable General Purpose Registers (GPRs), Control registers, Debug registers, or XMM registers. • The maximum width of each GPR is 32-bits limiting the amount of data each can hold. • Virtual memory address space available for each application is limited to 4GB by the 32-bit width of the linear (i.e., virtual) address. • The virtual-to-physical memory address translation mechanism limits the maximum addressable physical memory address space to 64GB. • The 32-bit Extended Instruction Pointer (EIP) register limits each application's code space to 4GB. • The x86 family's segmented memory model is complex and difficult to work with. Virtually all of today's OSs utilize a Flat Memory Model that effectively disables the segmented memory model. • The hardware-assisted task switching mechanism defined by the IA-32 ISA is slow and cumbersome. • IA-32 Mode permits virus code to be loaded into a stack or data segment from which it can then be executed. • Lacks the ability to address code-local data by specifying an address relative to the current EIP value.

Table 2-1: Basic Execution Modes (Continued)

Mode	Description
IA-32e Mode	IA-32 Extended Mode is comprised of two submodes: – 64-bit Mode. – Compatibility Mode. • Must be enabled by a 64-bit capable OS. • Provides an environment for the execution of 64-bit applications, as well as existing 32- and 16-bit Protected Mode applications. • Doesn't support the execution of VM86 Mode applications (i.e., MS-DOS applications). • Provides a fast transition between a 32-bit environment (Compatibility Mode) and a 64-bit environment (64-bit Mode). • Implements the Intel 64 extensions (formerly known as x86-64 or EM64T). "IA-32e SubModes" on page 28 describes the IA-32e execution Sub-Modes. **Some benefits associated with IA-32e Mode**: The following are some of the benefits realized when the logical processor is executing in IA-32e Mode: • Backward compatible with the IA-32 code environment. Intel's earlier attempt at a 64-bit architecture (Itanium) is not. • Expands the size of the virtual memory address space from 2^{32} (4GB) to 2^{64} (16EB; EB = exabytes). • Expands the size of the physical memory address space to 2^{52} (4PB; PB = petabytes). • The larger number of data registers permits a greater number of data variables to be accessed / manipulated rapidly: – Yields faster data set accessibility. – Widening and increasing the number of registers diminishes the number of accesses to memory and translates into improved performance. – The degree of improvement in kernel code efficiency depends on a kernel rewrite to manage memory better and to utilize 64-bit (rather than 32-bit) data variables. – The degree of improvement in application code efficiency depends on utilization of 64-bit data variables and, for large scale applications, capitalizing on the enlarged virtual address space.

3 A (very) Brief History

The Previous Chapter

The previous chapter introduced the execution modes and submodes and mode switching basics.

This Chapter

This chapter introduces the evolution of the x86 ISA, as well as the basic operational characteristics of 8086 Real Mode, 286 Protected Mode, and 386 Protected Mode. It also introduces the Intel microarchitecture families including a product introduction timeline.

The Next Chapter

The next chapter defines the state of a logical processor immediately after the removal of reset and introduces the concept of a soft reset (also referred to as an INIT). It also describes the initial code fetches performed by the BootStrap Processor as well as the methodology utilized by software to discover and configure all of the logical processors in the system.

Major Evolutionary Developments

Through the years, the x86 software architecture has steadily evolved with the introduction of new x86 processors. Some changes were small, evolutionary ones while others made significant additions to the architecture. Figure 3-1 on page 42 illustrates (and Table 3-1 on page 43 describes) those that, in the author's opinion, fall into the latter category.

Figure 3-1: Major Milestones in Evolution of Software Environment

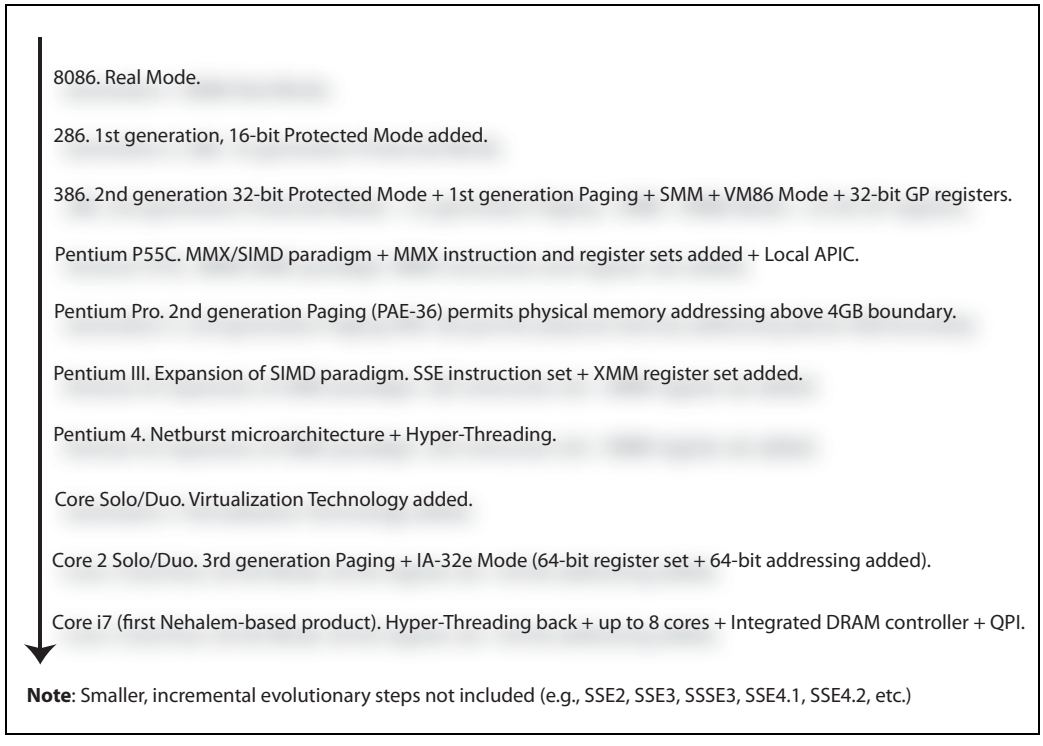


Table 3-1: Major Evolutionary Developments

Introduced In	Major Enhancements to the x86 Instruction Set Architecture
8086	At the root of the x86 family tree lies the 8086, a processor with the following characteristics: • Modes: **Real Mode**. • Addressable Memory: 1MB. • Data Transfer Width: 16-bits. • Programming Model: 16-bit.
286	Next, we have the 286, a processor with the following characteristics: • Modes: Real Mode and **first generation 16-bit Protected Mode.** • Missing three critical capabilities: – It did not implement a virtual-to-physical address translation facility (i.e., the Paging mechanism). – Could not address more than 16MB of physical memory. – It did not implement Virtual 8086 (VM86) Mode, an exemption that effectively crippled the processor's ability to run ill-behaved DOS applications under a multi-tasking OS. • Addressable Physical Memory: 16MB. • Data Transfer Width: 16-bits. • Programming Model: 16-bit (16-bit GP registers and 16-bit addressing).

Table 3-1: Major Evolutionary Developments (Continued)

Introduced In	Major Enhancements to the x86 Instruction Set Architecture
386	The introduction of the 386 contributed the following major architectural changes: • **2nd generation 32-bit Protected Mode**. Permits 32-bit addressing (rather than the 16-bit addressing supported by the 286's first generation Protected Mode). • **1st generation virtual-to-physical address translation** mechanism (i.e., Paging). Permitted a 32-bit virtual memory address to be translated into a 32-bit physical memory address. • **System Management Mode (SMM)** first appeared in the 386SX processor. If the platform logic detects a platform-specific issue (e.g., a thermal zone is warming up), the chipset generates an SM Interrupt (SMI) to the processor which interrupts the currently-running program, saves the processor's register set and executes the SMM handler. The handler checks chipset status to determine the nature of the problem, handles the problem (e.g., by turning on a fan) and then restores the processor's register set and resumes execution of the interrupted program. • **VM86 Mode**. This mechanism permits the processor hardware to monitor the execution of ill-behaved DOS applications on an instruction-by-instruction basis. If an instruction that could destabilize the multi-tasking OS is detected, the DOS program is interrupted and a special program, the VMM (Virtual Machine Monitor), is executed to determine the nature of the problem and fix it. The OS then resumes execution of the DOS application. • Addressable Physical Memory: 4GB. • Data Transfer Width: 32-bits. • Programming Model: 32-bit (32-bit GP registers and 32-bit addressing).

4 State After Reset

The Previous Chapter

The previous chapter introduced the evolution of the x86 ISA, as well as the basic operational characteristics of 8086 Real Mode, 286 Protected Mode, and 386 Protected Mode. It also introduced the Intel microarchitecture families including a product introduction timeline.

This Chapter

This chapter defines the state of a logical processor immediately after the removal of reset and introduces the concept of a soft reset (also referred to as an INIT). It also describes the initial code fetches performed by the BootStrap Processor and introduces the methodology utilized by software to discover and configure all of the logical processors in the system.

The Next Chapter

The next chapter provides a very basic introduction to the various facilities that support the IA-32 computing environment. These facilities include:

- Pre-386 Register Sets (this section is provided for historical background).
- IA-32 Register Set Overview.
- Control Registers.
- Status / Control Register (Eflags).
- Instruction Fetch Facilities.
- General Purpose Data Registers.
- Defining Memory Regions / Characteristics.
- Interrupt / Exception Facilities.
- Kernel Facilities.
- Address Translation Facilities.
- Legacy FP Facilities.
- MMX Facilities.

- SSE Facilities.
- Model-Specific Registers.
- Debug Facilities.
- Automatic Task Switching Mechanism.

State After Reset

Table 4-1 on page 66 defines the state of the logical processor's registers (the IA-32 register set is shown in Figure 4-1 on page 65) and resources immediately after the removal of reset. To summarize:

- The logical processor is in Real Mode (Protected Mode and Paging are disabled).
- Its caches are empty and caching is disabled.
- All of the feature bits in CR4 are cleared disabling most of the new features introduced after the advent of the 386.
- Recognition of external hardware interrupts is disabled.
- No instructions have been fetched from memory.
- The x87 FPU is disabled.
- All x87 FPU and SSE exceptions are disabled.
- The Machine Check and Alignment Check exceptions are disabled.
- The first instruction will be fetched from location FFFFFFF0h.

Figure 4-1: IA-32 Register Set

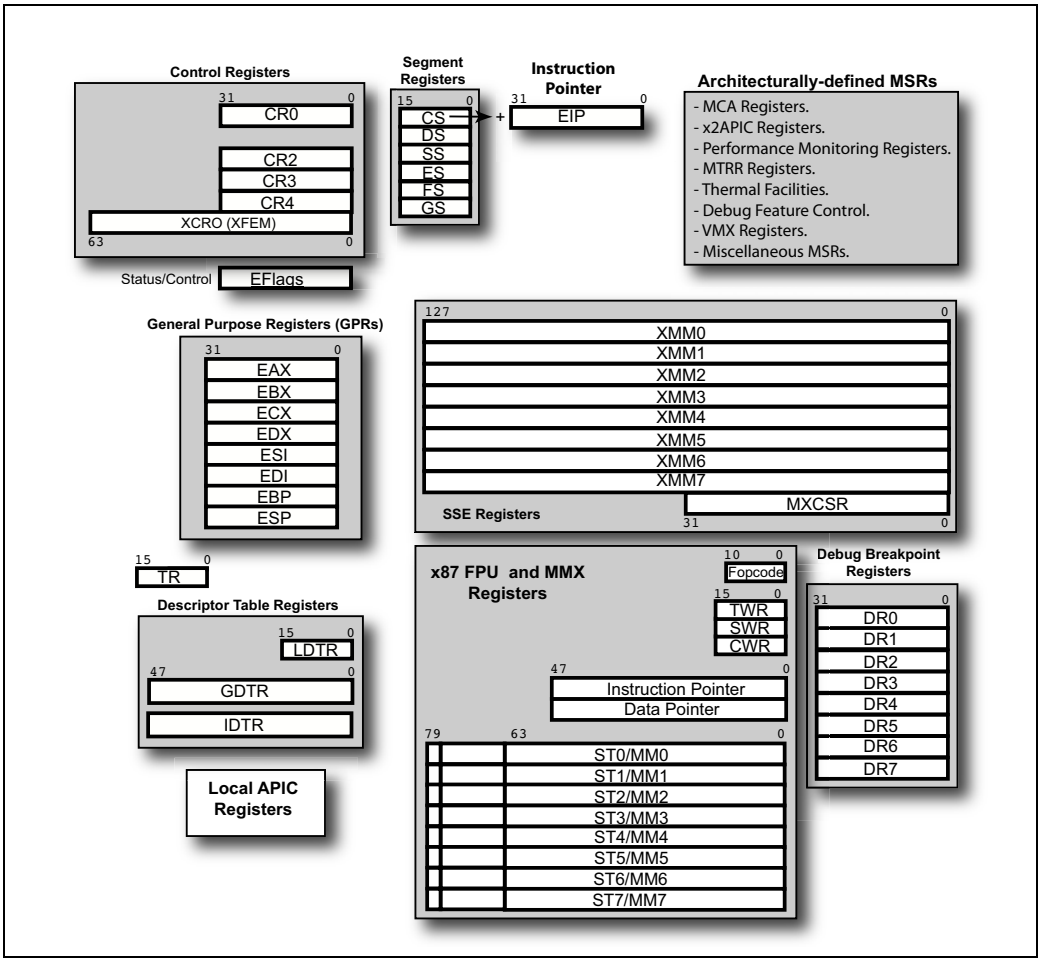


Table 4-1: Logical Processor State After Removal of Reset

Register or Resource	Effect(s)
Instruction Pipeline	No instructions have been fetched from memory yet.
ROB	Since no instructions have been fetched from memory yet to be translated into micro-ops, the Reorder Buffer is empty and the instruction dispatch logic is idle.
BTB Cache	The Branch Target Buffer maintains history on branch execution (i.e., whether branches were taken or not taken) is empty.
CR0 register	Contains 60000010h after reset: • CR0[PE] = 0, disabling Protected Mode. The logical processor is therefore in Real Mode. • CR0[PG] = 0, disabling Paging (virtual-to-physical address translation services). • CR0[CD&NW] = 11b, disabling the caching logic. • CR0[EM&MP] = 11b, indicating the x87 FPU isn't present and the logical processor should therefore permit software to emulate it (by executing integer-only code). • CR0[TS] = 0, indicating a task switch has not occurred. • CR0[ET] = 1, indicating that the integrated x87 FPU is compatible with the 387 FPU. • CR0[NE] = 0, disabling the logical processor's ability to generate an exception 16 if an x87 FP exception occurs. Instead, the logical processor signals the event using the DOS-compatible method (i.e., by asserting the processor's FERR# output which causes the assertion of IRQ13 to the 8259A PIC (Programmable Interrupt Controller). • CR0[WP] = 0. This has no effect at this time because Paging is disabled. When paging is enabled, setting this bit to a one prevents privilege level 0 software from writing to read-only pages. • CR0[AM] = 0. Clearing the Alignment Mask bit disables the logical processor's ability to generate the Alignment Check exception when a mis-aligned multi-byte memory access is detected.

5 Intro to the IA-32 Ecosystem

The Previous Chapter

The previous chapter defined the state of a logical processor immediately after the removal of reset and introduced the concept of a soft reset (also referred to as an INIT). It also described the initial code fetches performed by the BootStrap Processor and introduced the methodology utilized by software to discover and configure all of the logical processors in the system.

This Chapter

This chapter provides a very basic introduction to the various facilities that support the IA-32 computing environment. These facilities include:

- Pre-386 Register Sets (this section is provided for historical background).
- IA-32 Register Set Overview.
- Control Registers.
- Status / Control Register (Eflags).
- Instruction Fetch Facilities.
- General Purpose Data Registers.
- Defining Memory Regions / Characteristics.
- Interrupt / Exception Facilities.
- Kernel Facilities.
- Address Translation Facilities.
- Legacy FP Facilities.
- MMX Facilities.
- SSE Facilities.
- Model-Specific Registers.
- Debug Facilities.
- Automatic Task Switching Mechanism.

The Next Chapter

The next chapter highlights the dramatic expansion of the x86 instruction set since the advent of the 386 by listing both the 386 instruction set as well as the current-day instruction set.

The Pre-386 Register Sets

This section provides a little background regarding the baseline register set that was a precursor to the expanded register set found in today's x86 processors. Basic descriptions of these registers are included in this chapter.

8086 Register Set

The 8086 register set (see Figure 5-1 on page 80) consisted of the following registers:

- Eight General Purpose Registers (GPRs). See Figure 5-2 on page 81.
- Flags register. See Figure 5-3 on page 81.
- Four Segment registers.
- Instruction Pointer (IP) register.

Figure 5-1: 8086 Register Set

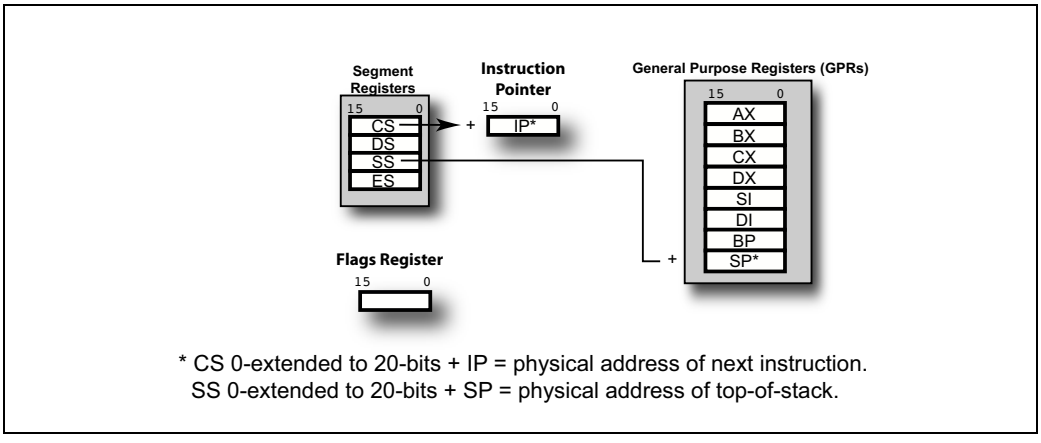


Figure 5-2: 8086 GPRs

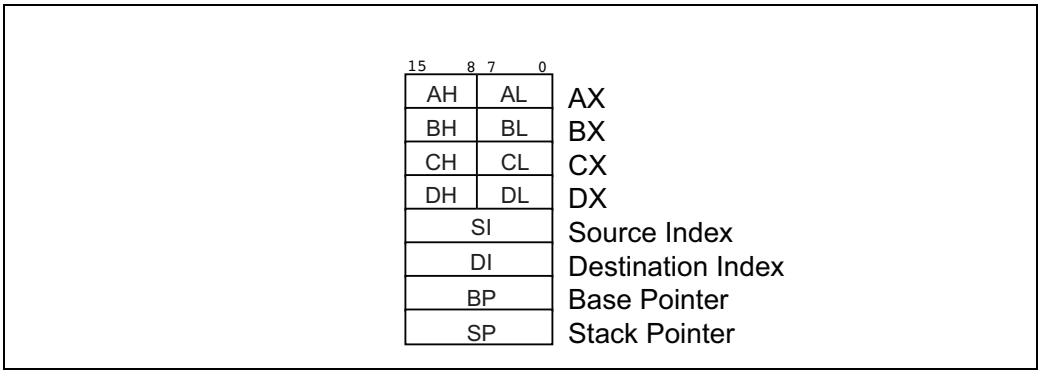
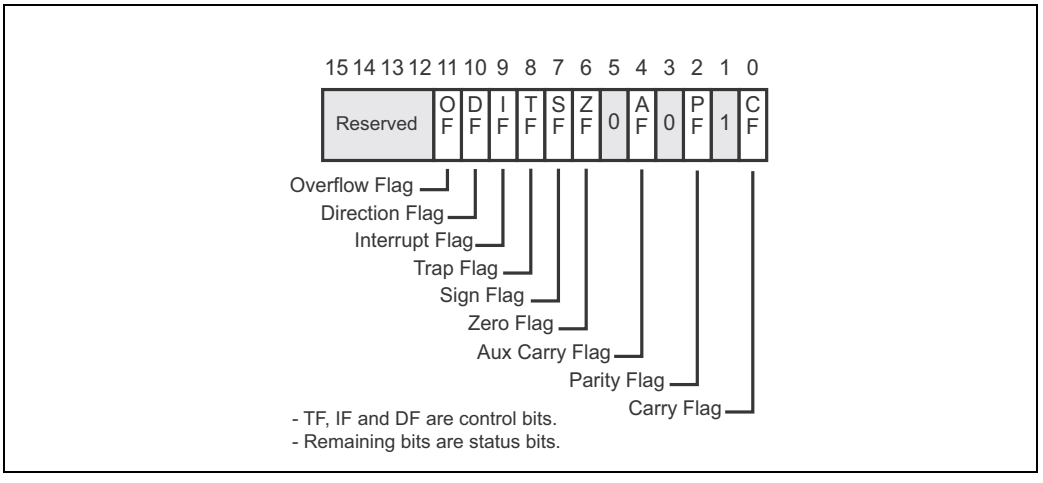


Figure 5-3: 8086 Flag Register



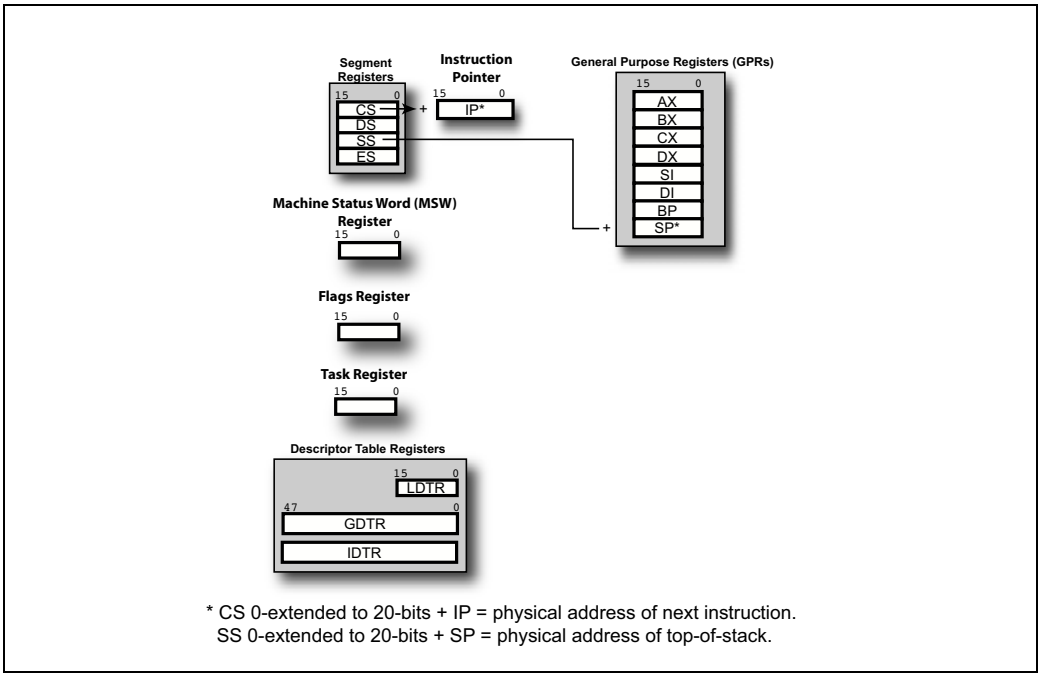
286 Register Set

The 286 added the following registers (see Figure 5-4 on page 82):

- Machine Status Word (MSW) register. See Figure 5-5 on page 83.
- Kernel-related registers:
  - Task Register (TR).
  - Interrupt Descriptor Table Register (IDTR).
  - Global Descriptor Table Register (GDTR).
  - Local Descriptor Table Register (LDTR).

It also added two additional bit fields (Nested Task and IO Privilege Level) to the Flags register (see Figure 5-6 on page 83).

Figure 5-4: 286 Register Set



6 Instruction Set Expansion

The Previous Chapter

The previous chapter provided a very basic introduction to the various facilities that support the IA-32 computing environment. These facilities include:

- Pre-386 Register Sets (this section is provided for historical background).
- IA-32 Register Set Overview.
- Control Registers.
- Status / Control Register (Eflags).
- Instruction Fetch Facilities.
- General Purpose Data Registers.
- Defining Memory Regions / Characteristics.
- Interrupt / Exception Facilities.
- Kernel Facilities.
- Address Translation Facilities.
- Legacy FP Facilities.
- MMX Facilities.
- SSE Facilities.
- Model-Specific Registers.
- Debug Facilities.
- Automatic Task Switching Mechanism.

This Chapter

This chapter illustrates the expansion of the x86 instruction set since the advent of the 386 by listing both the 386 instruction set as well as the current-day instruction set.

The Next Chapter

The next chapter provides a detailed explanation of the structure of an IA-32 instruction and covers the following topics:

- Effective Operand Size.
- Instruction Composition.
- Instruction Format Basics.
- Opcode (Instruction Identification).
  - In the Beginning.
  - 1-byte Opcodes.
  - 2-byte Opcodes Use 2-Level Lookup.
  - 3-byte Opcodes Use 3-Level Lookup.
  - Opcode Micro-Maps (Groups).
  - x87 FP Opcodes Inhabit Opcode Mini-Maps.
  - Special Opcode Fields.
- Operand Identification.
  - Specifying Registers as Operands.
  - Addressing a Memory-Based Operand.
  - Specifying an Immediate Value As an Operand.
- Instruction Prefixes.
  - Operand Size Override Prefix (66h).
  - Address Size Override Prefix (67h).
  - Lock Prefix.
  - Repeat Prefixes.
  - Segment Override Prefix.
  - Branch Hint Prefix.
- Summary of Instruction Set Formats.

Why a Comprehensive Instruction Set Listing Isn't Included

Since the Intel and AMD x86 instruction set reference guides already do a fine job fulfilling this role, this chapter does not provide a comprehensive description of each instruction in the x86 instruction set. Rather, it is intended as an introduction to the instruction set. To lend historical perspective, it begins with a listing of the entire 386 instruction set (all 128 of them) and then continues with a listing of the current instruction set (well over 400 instructions as of March 2009) sorted by category. It should be stressed that the instruction set is constantly evolving, maintaining backward-compatibility even as successive generations of x86 processors continue to add new instructions—sometimes in

small numbers; at other times, with substantial additions to the instruction repertoire (e.g., MMX and SSE).

386 Instruction Set

To lend historical perspective, Table 6-1 on page 111 lists the 386 processor's instruction set organized by category.

Table 6-1: 386 Instruction Set

Instruction	Description
Data Transfer—General Purpose	
MOV	Move operand
PUSH	Push operand onto stack
POP	Pop operand off stack
PUSHA	Push all registers on stack
POPA	Pop all registers off stack
XCHG	Exchange Operand, Register
XLAT	Translate
Data Transfer—Conversion	
MOVZX	Move byte or Word, DW, with zero extension
MOVSX	Move byte or Word, DW, sign extended
CBW	Convert byte to Word, or Word to DW
CWD	Convert Word to DW
CWDE	Convert Word to DW extended
CDQ	Convert DW to QW
Data Transfer—Input/Output	
IN	Input operand from I/O space

Table 6-1: 386 Instruction Set (Continued)

Instruction	Description
OUT	Output operand to I/O space
Data Transfer—Address Object	
LEA	Load effective address
LDS	Load pointer into D segment register
LES	Load pointer into E segment register
LFS	Load pointer into F segment register
LGS	Load pointer into G segment register
LSS	Load pointer into S (Stack) segment register
Data Transfer—Flag Manipulation	
LAHF	Load A register from Flags
SAHF	Store A register in Flags
PUSHF	Push flags onto stack
POPF	Pop flags off stack
PUSHFD	Push Eflags onto stack
POPFD	Pop Eflags off stack
CLC	Clear Carry Flag
CLD	Clear Direction Flag
CMC	Complement Carry Flag
STC	Set Carry Flag
STD	Set Direction Flag
Arithmetic Instructions - Addition	
ADD	Add operands
ADC	Add with carry

7 32-bit Machine Language Instruction Format

The Previous Chapter

The previous chapter illustrated the expansion of the x86 instruction set since the advent of the 386 by listing both the 386 instruction set as well as the current-day instruction set.

This Chapter

This chapter provides a detailed explanation of the structure of an IA-32 instruction and covers the following topics:

- Effective Operand Size.
- Instruction Composition.
- Instruction Format Basics.
- Opcode (Instruction Identification).
  - In the Beginning.
  - 1-byte Opcodes.
  - 2-byte Opcodes Use 2-Level Lookup.
  - 3-byte Opcodes Use 3-Level Lookup.
  - Opcode Micro-Maps (Groups).
  - x87 FP Opcodes Inhabit Opcode Mini-Maps.
  - Special Opcode Fields.
- Operand Identification.
  - Specifying Registers as Operands.
  - Addressing a Memory-Based Operand.
  - Specifying an Immediate Value As an Operand.
- Instruction Prefixes.
  - Operand Size Override Prefix (66h).
  - Address Size Override Prefix (67h).
  - Lock Prefix.
  - Repeat Prefixes.

  - Segment Override Prefix.
  - Branch Hint Prefix.
- Summary of Instruction Set Formats.

The Next Chapter

The next chapter provides a detailed description of Real Mode operation and covers the following topics:

- 8086 Emulation.
- Unused Facilities.
- Real Mode OS Environment.
- Running Real Mode Applications Under a Protected Mode OS.
- Real Mode Applications Aren't Supported in IA-32e Mode.
- Real Mode Register Set.
- IO Space versus Memory Space.
- IO and Memory-Mapped IO Operations.
- Operand Size Selection.
- Address Size Selection.
- Real Mode Memory Addressing.
- Real Mode Interrupt/Exception Handling.
- Summary of Real Mode Limitations.

64-bit Machine Language Instruction Format

As its name implies, the current chapter provides a detailed description of the 32-bit machine language instruction format. The 64-bit extensions to the machine language instruction format are covered in:

- "64-bit Operands and Addressing" on page 1041.
- "64-bit Odds and Ends" on page 1075.

A Complex Instruction Set with Roots in the Past

As mentioned earlier in "IA Instructions vs. Micro-ops" on page 15, the x86 machine language instruction set is quite complex. Depending on the type of instruction, the number of operands it specifies, and the operand types (memory- and/or register-based), a single instruction may consist of anywhere between one and fifteen bytes. Beginning with the advent of the Pentium Pro processor, all x86 processors incorporate a translator that converts each IA-32

machine language instruction into a series of one or more simple, fixed-length micro-ops which are then executed by the logical processor.

Effective Operand Size

Introduction

In order to limit the number of opcodes, the same opcode is used for an instruction whether it operates on an 8-, 16- or 32-bit operand. As an example:

```
mov ax,dx
mov eax,edx
```

both use the same basic opcode. This naturally brings up a question: how, then, does the logical processor determine which registers are being referenced? The answer is simple and is described in the next two sections.

Operand Size in 16- and 32-bit Code Segments

Assuming that an instruction is not prefaced by a Operand Size Override prefix byte (66h), the logical processor behaves as outlined in Table 7-1 on page 157. Adding the prefix byte before the instruction's first opcode byte alters its behavior as defined in Table 7-2 on page 158.

Table 7-1: Effective Operand Size in 16- or 32-bit Mode (without prefix)

State of D-bit in active CS Descriptor	Effective Operand Size and Instruction Behavior
0: 16-bit, 286 CS descriptor.	**16-bits**. Instruction operates on 16-bits (a word) in a 16-bit register (e.g., AX) and either of the following: • Two sequential memory locations. • Another 16-bit register.
1: 32-bit, 386 CS descriptor.	**32-bits**. Instruction operates on 32-bits (a dword) in a 32-bit register (e.g., EAX) and either of the following: • Four sequential memory locations. • Another 32-bit register.

Table 7-1: Effective Operand Size in 16- or 32-bit Mode (without prefix) (Continued)

State of D-bit in active CS Descriptor	Effective Operand Size and Instruction Behavior
For some instructions, the opcode contains a width (W) bit: • W = 0. The operand size is 8- rather than 16- or 32-bits. • W = 1. The operand size is either 16- or 32-bits (based on the state of CSDesc[D] and the presence or absence of the Operand Size Override prefix).	

Table 7-2: Effective Operand Size in 16- or 32-bit Mode (with prefix)

State of D-bit in active CS Descriptor	Effective Operand Size and Instruction Behavior
0: 16-bit, 286 CS descriptor.	**Inclusion of the Operand Size Override prefix before the instruction flips the effective operand size from 16- to 32-bits**. Instruction operates on 32-bits (a dword) in a 32-bit register (e.g., EAX) and either of the following: • Four sequential memory locations. • Another 32-bit register.
1: 32-bit, 386 CS descriptor.	**Inclusion of the Operand Size Override prefix before the instruction flips the effective operand size from 32- to 16-bits**. Instruction operates on 16-bits (a word) in a 16-bit register (e.g., AX) and either of the following: • Two sequential memory locations. • Another 16-bit register.
For some instructions, the opcode contains a width (W) bit: • W = 0. The operand size is 8- rather than 16- or 32-bits. • W = 1. The operand size is either 16- or 32-bits (based on the state of CSDesc[D] and the presence or absence of the Operand Size Override prefix).	

Operand Size in 64-bit Code Segments

The default data operand size when executing code from a 64-bit code segment (code segment descriptor's L bit = 1) is 32-bits and its default address size is 64-bits. In other words, unless instructed otherwise, the logical processor assumes

8 Real Mode (8086 Emulation)

The Previous Chapter

The previous chapter provided a detailed explanation of the structure of an IA-32 instruction and covered the following topics:

- Effective Operand Size.
- Instruction Composition.
- Instruction Format Basics.
- Opcode (Instruction Identification).
  - In the Beginning.
  - 1-byte Opcodes.
  - 2-byte Opcodes Use 2-Level Lookup.
  - 3-byte Opcodes Use 3-Level Lookup.
  - Opcode Micro-Maps (Groups).
  - x87 FP Opcodes Inhabit Opcode Mini-Maps.
  - Special Opcode Fields.
- Operand Identification.
  - Specifying Registers as Operands.
  - Addressing a Memory-Based Operand.
  - Specifying an Immediate Value As an Operand.
- Instruction Prefixes.
  - Operand Size Override Prefix (66h).
  - Address Size Override Prefix (67h).
  - Lock Prefix.
  - Repeat Prefixes.
  - Segment Override Prefix.
  - Branch Hint Prefix.
- Summary of Instruction Set Formats.

This Chapter

This chapter provides a detailed description of Real Mode operation and covers the following topics:

- 8086 Emulation.
- Unused Facilities.
- Real Mode OS Environment.
- Running Real Mode Applications Under a Protected Mode OS.
- Real Mode Applications Aren't Supported in IA-32e Mode.
- Real Mode Register Set.
- IO Space versus Memory Space.
- IO and Memory-Mapped IO Operations.
- Operand Size Selection.
- Address Size Selection.
- Real Mode Memory Addressing.
- Real Mode Interrupt/Exception Handling.
- Summary of Real Mode Limitations.

The Next Chapter

The next chapter provides a detailed description of the x87 FPU and covers the following topics:

- A Little History.
- x87 FP Instruction Format.
- FPU-Related CR0 Bit Fields.
- x87 FPU Register Set.
  - The FP Data Registers.
  - x87 FPU's Native Data Operand Format.
  - 32-bit SP FP Numeric Format.
  - DP FP Number Representation.
  - FCW Register.
  - FSW Register.
  - FTW Register.
  - Instruction Pointer Register.
  - Data Pointer Register.
  - Fopcode Register.
- FP Error Reporting.
  - Precise Error Reporting.
  - Imprecise (Deferred) Error Reporting.
  - Why Deferred Error Reporting Is Used.
  - The WAIT/FWAIT Instruction.
  - CR0[NE].
  - Ignoring FP Errors.

8086 Emulation

Real Mode was introduced with the advent of the 8086/8088 processors, and, due to the huge success of the IBM PC and the proliferation of software written for the Real Mode environment, Intel could ill-afford to leave it behind. Immediately after the removal of reset, all subsequent x86 processors emulate the operation of the 8086 by initiating operation in Real Mode. There are, of course, some differences (see Table 8-1 on page 229).

Table 8-1: Basic Differences Between 8086 Operation and Real Mode

Difference	Basic Description
Speed of execution	In the IBM PC, the processor clock ran at 4.77MHz. Today's processors execute Real Mode code hundreds of times faster.
Cache boost	Today's processors enjoy a substantial performance boost due to on-chip caches.
Accessing extended memory	While the 8086/8088 processors were strictly limited to a 1MB address space due to an address bus width of 20-bits, current-day x86 processors can, even in Real Mode, access significantly more memory: • See "Accessing Extended Memory in Real Mode" on page 307. • See "Big Real Mode" on page 310.
Operand and address size	**Operand Size**. While the default data operand size in Real Mode is 16-bits, 32-bit operands can be specified by prefacing an instruction with the Operand Size Override prefix. The logical processor can then access 32-bit operands in memory as well as the 32-bit GPR registers: EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI. The 8086/8088 GPR registers were only 16-bits wide. **Address Size**. While the default address size for memory-based operands in Real Mode is 16-bits, a 32-bit address can be specified by prefacing an instruction with the Address Size Override prefix.

Table 8-1: Basic Differences Between 8086 Operation and Real Mode (Continued)

Difference	Basic Description
Number of memory data segments	Using the ES, FS and GS Segment Override prefixes, the programmer can access memory-based data operands in the E, F and G data segments. The 8086/8088 only implemented the DS data segment.
Integrated x87 FPU	Today's logical processors incorporate an integrated x87 FPU. The 8087 FPU was implemented as an optional, external companion device to the 8086/8088. Upon detection of a FP instruction, the processor had to forward the instruction to the x87 FPU by performing a series of IO write transactions on its external interface. Very slow, indeed.
Debug register set	The address breakpoint facility is available through the Debug register set (not implemented on the 8086/8088).
Additional instructions available	The instructions listed in Table 8-2 on page 230, although not available on the 8086/8088, can be used in Real Mode.

Table 8-2: Expanded/Enhanced Real Mode Instructions

Instructions Available (that weren't present in the 8086/8088)
The MMX instruction set as well as the MMX data registers (MM0 - MM7).
The SSE1, SSE2, SSE3, SSSE3, SSE4.1 and SSE4.2 instruction sets as well as the SSE register set and the SSE FP exception (exception 19).
MOV instructions that operate on the Control and Debug registers.
Load segment register instructions: LSS, LFS, and LGS.
Generalized multiply and multiply immediate data instructions.
Shift and rotate by immediate counts.
PUSHA, PUSHAD, POPA and POPAD, and PUSH immediate data stack instructions.
MOVSX and MOVZX Move with sign extension instructions.

9 Legacy x87 FP Support

The Previous Chapter

The previous chapter provided a detailed description of Real Mode operation and covered the following topics:

- 8086 Emulation.
- Unused Facilities.
- Real Mode OS Environment.
- Running Real Mode Applications Under a Protected Mode OS.
- Real Mode Applications Aren't Supported in IA-32e Mode.
- Real Mode Register Set.
- IO Space versus Memory Space.
- IO and Memory-Mapped IO Operations.
- Operand Size Selection.
- Address Size Selection.
- Real Mode Memory Addressing.
- Real Mode Interrupt/Exception Handling.
- Summary of Real Mode Limitations.

This Chapter

This chapter provides a detailed description of the x87 FPU and covers the following topics:

- A Little History.
- x87 FP Instruction Format.
- FPU-Related CR0 Bit Fields.
- x87 FPU Register Set.
  - The FP Data Registers.
  - x87 FPU's Native Data Operand Format.

  - 32-bit SP FP Numeric Format.
  - DP FP Number Representation.
  - FCW Register.
  - FSW Register.
  - FTW Register.
  - Instruction Pointer Register.
  - Data Pointer Register.
  - Fopcode Register.
- FP Error Reporting.
  - Precise Error Reporting.
  - Imprecise (Deferred) Error Reporting.
  - Why Deferred Error Reporting Is Used.
  - The WAIT / FWAIT Instruction.
  - CR0[NE].
  - Ignoring FP Errors.

The Next Chapter

The next chapter provides an introduction to the concept of multitasking and covers the following topics:

- Concept.
- An Example—Timeslicing.
- Another Example—Awaiting an Event.
  - 1. Task Issues Call to OS for Disk Read.
  - 2. Device Driver Initiates Disk Read.
  - 3. OS Suspends Task.
  - 4. OS Makes Entry in Event Queue.
  - 5. OS Starts or Resumes Another Task.
  - 6. Disk-Generated Interrupt Causes Jump to OS.
  - 7. Interrupted Task Suspended.
  - 8. Task Queue Checked.
  - 9. OS Resumes Task.

A Little History

Prior to the advent of the 486DX processor, x86 processors did not include an on-die FPU. In order to perform floating-point (FP) math operations the end user had to add an external x87 FPU chip to the system which the processor treated as a specialized IO device. When the processor encountered a FP instruction in the currently-executing program, it would perform a series of one or more IO write transactions on its external bus to forward the instruction to the off-chip FPU for execution. Obviously, this was very inefficient.

The 486DX was the first x86 processor to integrate the x87 FPU (all subsequent x86 processors include it). The sections that follow provide a description of the FPU's register set and the format in which FP numbers are represented.

x87 FP Instruction Format

This topic is covered in "x87 FP Opcodes Inhabit Opcode Mini-Maps" on page 187.

FPU-Related CR0 Bit Fields

Refer to Figure 9-1 on page 341, Table 9-1 on page 341, and Table 8-4 on page 241 for a description of the CR0 bit fields related to the x87 FPU.

Figure 9-1: CR0

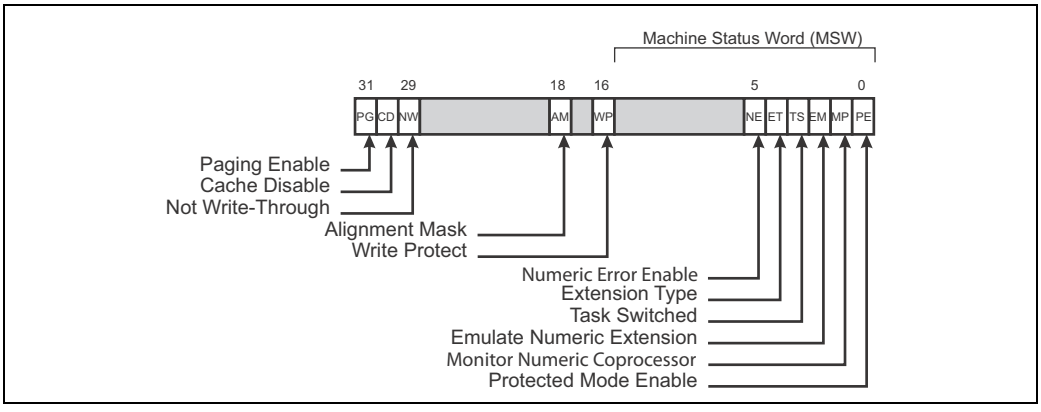


Table 9-1: CR0 x87 FPU-related Bit Fields

Bit(s)	Name	Description
2:1	EM, MP	The OS uses these x87 FPU-related bits to indicate whether the logical processor is: • **Running DOS**. • **Running a multitasking OS**. • **Neither** of the above (the FPU isn't present or is disabled); in this case, a software exception is generated when the logical processor detects an x87 instruction and software **emulates the x87 FPU.** See Table 8-4 on page 241.

Table 9-1: CR0 x87 FPU-related Bit Fields (Continued)

Bit(s)	Name	Description
3	TS	**x87 Task Switch status bit**. The x87 FPU registers are not saved on an automatic task switch. When a hardware-based task switch occurs: • CR0[TS] is set to one. • Most but not all registers are automatically saved in the TSS (Task State Segment) data structure associated with the currently-running task. • When an attempt is made to execute an x87 FP instruction or an MMX instruction while CR0[TS] = 1, a DNA (Device Not Available) Exception 7 is generated. • The DNA exception handler executes FSAVE or FXSAVE to save the x87 and, possibly, the SSE registers in the TSS of the task that last used the x87. This information is saved in an OS-designated area of the previous task's TSS. • The DNA exception handler then clears CR0[TS] and executes an IRET instruction to return to the x87 instruction in the current task that caused the DNA exception. It now executes successfully.
4	ET	**x87 FPU type**. In processors prior to the 486DX, CR0[ET] (ET = Extension Type) was a read / write bit used by software to indicate the type of numeric coprocessor installed on the system board (287 or 387 FPU-compatible). Since the advent of the 486DX this bit is hardwired to one indicating that the logical processor incorporates a 387-style FPU.
5	NE	**Numeric Exception**. Controls whether FP errors are reported using the DOS-compatible method (IRQ13; see "DOS-Compatible FP Error Reporting" on page 359) or by generating an exception 16. The OS kernel sets NE = 1 if it incorporates an x87 FP exception handler. Any x87 FPU error then causes the logical processor to generate an internal exception 16 (rather than using the DOS-compatible method and asserting its external FERR# signal).

10 Introduction to Multitasking

The Previous Chapter

The previous chapter provided a detailed description of the x87 FPU and covered the following topics:

- A Little History.
- x87 FP Instruction Format.
- FPU-Related CR0 Bit Fields.
- x87 FPU Register Set.
  - The FP Data Registers.
  - x87 FPU's Native Data Operand Format.
  - 32-bit SP FP Numeric Format.
  - DP FP Number Representation.
  - FCW Register.
  - FSW Register.
  - FTW Register.
  - Instruction Pointer Register.
  - Data Pointer Register.
  - Fopcode Register.
- FP Error Reporting.
  - Precise Error Reporting.
  - Imprecise (Deferred) Error Reporting.
  - Why Deferred Error Reporting Is Used.
  - The WAIT/FWAIT Instruction.
  - CR0[NE].
  - Ignoring FP Errors.

This Chapter

This chapter provides an introduction to the concept of multitasking and covers the following topics:

- Concept.
- An Example—Timeslicing.
- Another Example—Awaiting an Event.
  - 1. Task Issues Call to OS for Disk Read.
  - 2. Device Driver Initiates Disk Read.
  - 3. OS Suspends Task.
  - 4. OS Makes Entry in Event Queue.
  - 5. OS Starts or Resumes Another Task.
  - 6. Disk-Generated Interrupt Causes Jump to OS.
  - 7. Interrupted Task Suspended.
  - 8. Task Queue Checked.
  - 9. OS Resumes Task.

The Next Chapter

The next chapter introduces the concepts of hardware-based task switching, global and local memory, privilege checking, read/write protection, IO port protection, interrupt masking, and BIOS call interception. The following topics are covered:

- Hardware-based Task Switching Is Slow!
- Private (Local) and Global Memory.
- Preventing Unauthorized Use of OS Code.
- With Privilege Comes Access.
  - Program Privilege Level.
    - The CPL.
    - Calling One of Your Equals.
    - Calling a Procedure to Act as Your Surrogate.
  - Data Segment Protection.
    - Data Segment Privilege Level.
    - Read-Only Data Areas.
- Some Code Segments Contain Data, Others Don't.
- IO Port Anarchy.
- No Interrupts, Please!
- BIOS Calls.

Concept

A multitasking OS does not run multiple programs (i.e., tasks) simultaneously. In reality, it loads a task into memory, permits it to run for a while and then suspends it. The program is suspended by creating a snapshot, or image, containing the contents of all or many of the logical processor's registers in memory (frequently referred to as the processor context). If the OS *were* using the hardware-based task switching mechanism included in the IA-32 architecture, (most modern OSs do not) the logical processor would automatically store the image in a special data structure in memory referred to as a Task State Segment (TSS) by performing an automated series of memory writes. In other words, the state of the logical processor at the point of suspension would be saved in memory. In reality, most modern OSs save the register contents (under software control) in an OS-specific data structure (rather than the TSS) which the author will refer to as the Task Data Structure. Some of the TSS's functionality is, in fact, used, however (this will be covered later).

Having effectively saved a snapshot that indicates the point of suspension and the logical processor's state at that time, the logical processor could then initiate another task by loading it into memory and jumping to its entry point, or it could resume a previously-suspended task by reloading its register set from that task's Task Data Structure. Based on OS-specific criteria, the OS could at some point decide to suspend this task as well. As before, the state of the logical processor would be saved in memory (in the Task Data Structure) as a snapshot of the task's state at its point of suspension.

When the OS decides to resume a previously-suspended task, the logical processor's registers would be restored from the Task Data Structure under software control by performing a series of memory reads. The logical processor would then use the address pointer stored in the CS:EIP register pair to fetch the next instruction, thereby resuming program execution at the point where it had been suspended earlier.

The circumstances under which an OS decides to suspend a task is specific to the OS. It may simply use *timeslicing* wherein each task is permitted to execute for a fixed amount of time (e.g., 10ms). At the end of that period of time, the currently executing task is suspended and the next task in the queue is started or resumed. The OS might be designed to suspend the currently executing program when it requests something that is not immediately available (e.g., when it attempts an access to a page of information that resides on a mass storage device and is currently not in memory). It starts or resumes another task and, when the event previously requested by the now-suspended task occurs, typi-

cally signaled by an interrupt, the current task is interrupted and suspended and the previously suspended task resumed. This is commonly referred to as *preemptive multitasking*.

An Example—Timeslicing

Prior to starting or resuming execution of a task:

1. The OS *task scheduler* would initialize a hardware timer (typically, the timer incorporated within the Local APIC associated with the logical processor) to interrupt program execution after a defined period of time (e.g., 10ms).
2. The scheduler then causes the logical processor to initiate or resume execution of the task.
3. The logical processor proceeds to fetch and execute the instructions comprising the task for 10ms.
4. When the hardware timer expires it generates an interrupt, causing the logical processor to suspend execution of the currently executing task and to switch back to the OS's task scheduler.
5. The scheduler then determines which task to initiate or resume next.

Another Example—Awaiting an Event

1. Task Issues Call to OS for Disk Read

The application program calls the OS requesting that a block of data be read from a disk drive into memory. The OS then forwards the request to the disk driver, and, having done so, suspends the task and either starts or resumes another one.

Rather than awaiting the completion of the disk read, the OS scheduler would better utilize the machine's resources by suspending the task that originated the request and transferring control to another program so work can be accomplished while the disk operation is in progress.

2. Device Driver Initiates Disk Read

The driver issues a call to malloc (the OS's memory allocation manager) requesting the allocation of a memory buffer to hold the requested data. After

11 Multitasking-Related Issues

The Previous Chapter

The previous chapter provided an introduction to the concept of multitasking and covered the following topics:

- Concept.
- An Example—Timeslicing.
- Another Example—Awaiting an Event.
  - 1. Task Issues Call to OS for Disk Read.
  - 2. Device Driver Initiates Disk Read.
  - 3. OS Suspends Task.
  - 4. OS Makes Entry in Event Queue.
  - 5. OS Starts or Resumes Another Task.
  - 6. Disk-Generated Interrupt Causes Jump to OS.
  - 7. Interrupted Task Suspended.
  - 8. Task Queue Checked.
  - 9. OS Resumes Task.

This Chapter

This chapter introduces the concept of hardware-based task switching, global and local memory, privilege checking, read/write protection, IO port protection, interrupt masking, and BIOS call interception. The following topics are covered:

- Hardware-based Task Switching Is Slow!
- Private (Local) and Global Memory.
- Preventing Unauthorized Use of OS Code.

- With Privilege Comes Access.
  - Program Privilege Level.
    - The CPL.
    - Calling One of Your Equals.
    - Calling a Procedure to Act as Your Surrogate.
  - Data Segment Protection.
    - Data Segment Privilege Level.
    - Read-Only Data Areas.
- Some Code Segments Contain Data, Others Don't.
- IO Port Anarchy.
- No Interrupts, Please!
- BIOS Calls.

The Next Chapter

The next brief chapter summarizes various situations that can destabilize a multitasking OS environment and the x86 protection mechanisms that exist to address each of them.

Hardware-based Task Switching Is Slow!

The multitasking OS loads multiple tasks into different areas of memory and permits each to run for a slice of time. As described in the previous chapter, it permits a task to run for its assigned timeslice, suspends it, permits another task to run for a timeslice, suspends it, etc. If the OS is executing on a fast processor with fast access to memory, this task switching can be accomplished so quickly that all of the tasks *appear* to be executing simultaneously.

While the logical processor is executing a task, the OS kernel and all of the other dormant tasks are resident in memory. When each of the tasks (and the OS kernel's scheduler) were suspended earlier in time, the logical processor created a snapshot of its register image in memory at the moment of task suspension. In the IA-32 environment, the typical OS sets up a separate Task Data Structure for each task. If the OS designers had chosen to utilize the x86 processor's hardware-based task switching mechanism, the processor would automatically save its register set in and restore it from a task's TSS when suspending or resuming a task. *In fact, though, due to the inefficiency of this mechanism, no modern, mainstream OSs use the hardware-based task switch mechanism.* Rather, the OS task scheduler performs the register set save and restore in software using a task-specific data structure the author refers to as the Task Data Structure.

While x86 processors support the hardware-based mechanism in IA-32 Mode to ensure that any software that does use it will function correctly, *the hardware mechanism is not supported in IA-32e Mode.*

Private (Local) and Global Memory

The currently executing application is typically only aware of two entities—itself and the OS that manages it—and is unaware of the existence of any other tasks that, although partially or fully present in memory, are currently suspended. The currently executing application should only be permitted to access its own, private memory and, perhaps, one or more areas of memory that the OS has designated as globally-accessible by multiple applications to permit data and/or code sharing. If it were permitted to perform memory writes anywhere in memory, it is entirely probable that it will corrupt the code, stack or data areas of programs that are in memory but currently suspended. Consider what would happen when the OS resumes execution of a task that had been corrupted while in suspension. Its program and/or data would have been corrupted, causing it to behave unpredictably when it resumes execution. The OS must protect suspended tasks (including itself!) from the currently executing task. If it doesn't, multitasking will not work reliably.

When an application is loaded into memory, the OS memory manager designates certain areas of memory for its use:

- Some areas of memory are designated as private (i.e., local) to the application. These segments could be defined by entries (segment descriptors) in the application's Local Descriptor Table (LDT; see Figure 11-1 on page 373).
- The OS may also designate one or more areas of memory that are globally accessible by multiple applications (thereby permitting the sharing of data or code). These segments could be defined by entries in the Global Descriptor Table (GDT; Figure 11-1 on page 373).

In addition to defining the accessibility of memory areas using the logical processor's segmentation mechanism, the OS memory manager can also accomplish this using the virtual-to-physical address translation mechanism (i.e., Paging).

Preventing Unauthorized Use of OS Code

The OS maintains the integrity of the system. It manages all shared resources and decides what task will run next and for how long. It should be fairly obvious that the person in charge must have more authority (i.e., greater privileges) than the other tasks. It would be ill-conceived to permit normal tasks to access certain logical processor control registers, OS-related tables in memory, etc.

This form of protection can be accomplished in two ways: assignment of privilege levels to programs and assignment of ownership to areas of memory. IA-32 processors utilize both methods. There are four privilege levels:

- **Level zero**. Greatest amount of privilege. Assigned to the heart, or kernel, of the OS. It handles the task queues, memory management, etc.
- **Level one**. Typically assigned to OS services that provide services to the application programs and device drivers.
- **Level two**. Typically assigned to device drivers that the OS uses to communicate with peripheral devices.
- **Level three**. Least-privileged. Assigned to application programs.

The application program operates at the lowest privilege level (3) because its actions must be restricted. The OS kernel has the highest privilege level (0) so that it can accomplish its job of managing every aspect of the system. The integrity of the system would be compromised if an application program could call highly-privileged parts of the OS code to accomplish things it shouldn't be able to do. This implies that the logical processor must have some way of comparing the privilege level of the calling program to that of the program being called. To gain entry into the called program, the calling program's privilege level (CPL, or Current Privilege Level) must equal or exceed the privilege level of the program it is calling. IA-32 processors incorporate this feature.

With Privilege Comes Access

Privilege level 0 code has access to all of the logical processor’s facilities: it can execute any instruction, access any register, and access all memory data segments. Privilege level 3 code (i.e., application code), on the other hand, only has access to a subset of the instruction set and register set.

Program Privilege Level

The CPL

Refer to Figure 11-1 on page 373. When a far jump or far call is performed, the new value loaded into the 16-bit CS register selects a segment descriptor in either the GDT or the LDT. The 2-bit DPL (Descriptor Privilege Level) field in the selected code segment descriptor becomes the logical processor’s CPL (Current Privilege Level).

There are multiple cases where the OS wishes to restrict access to the code that resides in a code segment and they are introduced in the sections that follow.

12 Summary of the Protection Mechanisms

The Previous Chapter

The previous chapter introduced the concept of hardware-based task switching, global and local memory, privilege checking, read/write protection, IO port protection, interrupt masking, and BIOS call interception. The following topics were covered:

- Hardware-based Task Switching Is Slow!
- Private (Local) and Global Memory.
- Preventing Unauthorized Use of OS Code.
- With Privilege Comes Access.
  - Program Privilege Level.
    - The CPL.
    - Calling One of Your Equals.
    - Calling a Procedure to Act as Your Surrogate.
  - Data Segment Protection.
    - Data Segment Privilege Level.
    - Read-Only Data Areas.
- Some Code Segments Contain Data, Others Don't.
- IO Port Anarchy.
- No Interrupts, Please!
- BIOS Calls.

This Chapter

This chapter summarizes various situations that can destabilize a multitasking OS environment and the x86 protection mechanisms that exist to address each of them.

The Next Chapter

The next chapter introduces segment register usage in Protected Mode and the roles of segment descriptors, the GDT, the LDTs, the IDT, and the general segment descriptor format. It also introduces the concept of the flat memory model. The following topics are covered:

- Real Mode Segment Limitations.
- An Important Reminder: Segment Base + Offset = Virtual Address.
- Descriptor Contains Detailed Segment Description.
- Segment Register—Selects Descriptor Table and Entry.
- The Descriptor Tables.
- General Segment Descriptor Format.
- Goodbye to Segmentation.

Protection-Related Mechanisms

This chapter is not intended as a detailed discussion of the various protection mechanisms available in the x86 architecture. Rather, they've been collected here in one place for ease of reference and as an introduction to the various topics related to protection.

Some of the protection mechanisms were introduced in the previous chapter (privilege level assignment, local and global memory segments, data segment write protection, preventing data accesses to code segments, and guarding access to IO ports). All of the protection mechanisms are listed in Table 12-1 on page 379.

Table 12-1: Protection Mechanisms

Condition	Mechanism
Attempted access to a segment by a program with insufficient privilege	Some examples: • Currently-running program attempts to access a data segment with a higher privilege level (i.e., the data segment's descriptor[DPL] value is numerically less than the CPL of the program). • Currently-running program executes a far jump or a far call to a procedure in a code segment with a higher privilege level (e.g., a privilege level 3 program attempts to jump to or call a procedure in a privilege level 0 code segment). This would result in an exception.
Attempted write to a read-only data segment	The data segment descriptor's W bit = 0 indicating it is a read-only data segment. This would result in an exception.
Attempted data read from a code segment	The code segment descriptor's R bit = 0 indicating it's an execute-only code segment. This would result in an exception (if R = 1, then the code segment contains read-only data as well as code).
Out-of-range access to a segment	The offset address specified exceeds the segment size specified in the target segment's descriptor. This would result in an exception.
Attempted page access by an under-privileged program	A program with a privilege level of 3 attempted to access a page whose PTE[U/S] bit = 0. This would result in an exception. Note: U/S stands for User/Supervisor.

Table 12-1: Protection Mechanisms (Continued)

Condition	Mechanism
Attempted write to a read-only page	There are two possibilities: • A privilege level 3 program attempted to write to a page whose PTE[R / W] bit = 0 marking it as a read-only page. • A program with supervisor privileges (i.e., it has a privilege level of 0, 1 or 2) attempted to write to a user page (its PTE[U / S] bit = 1) with PTE[R / W] bit = 0 marking it as a read-only user page *and CR0[WP] = 1 (indicating supervisor programs are not permitted to write into read-only user pages).* This would result in an exception.
Access to an absent page	A virtual memory address selected a PTE with the Page Present bit (bit 0) = 0 indicating that the target physical page isn't currently in memory. This would result in an Page Fault exception.
Attempted direct access to an IO or memory-mapped IO port	• **Access to an IO port**: – **By a Protected Mode task (other than a VM86 task)**. Any attempt by a program (other than a VM86 task) with a privilege level numerically greater than the Eflags[IOPL] threshold to execute the **IN, INS, OUT and OUTS** instructions will trigger a General Protection exception. – **By a VM86 task**. When an IN, INS, OUT or OUTS instruction is executed, the logical processor uses the 16-bit IO port address to index into the IO Permission bit map in the task's TSS data structure. The state of the selected bit determines whether the IO instruction is executed or a General Protection exception is generated. • **Access to a memory-mapped IO port**. This form of protection can be provided by the virtual-to-physical address translation mechanism (i.e., Paging). The virtual addresses of memory-mapped IO ports could be grouped into a page and the virtual page address could select a PTE that indicates the page isn't present in memory. Any attempted access within the page would then result in a Page Fault exception.

13 Protected Mode Memory Addressing

The Previous Chapter

The previous chapter summarized various situations that can destabilize a multitasking OS environment and the x86 protection mechanisms that exist to address each of them.

This Chapter

This chapter introduces segment register usage in Protected Mode and the roles of segment descriptors, the GDT, the LDTs, the IDT, and the general segment descriptor format. It also introduces the concept of the flat memory model. The following topics are covered:

- Real Mode Segment Limitations.
- An Important Reminder: Segment Base + Offset = Virtual Address.
- Descriptor Contains Detailed Segment Description.
- Segment Register—Selects Descriptor Table and Entry.
- The Descriptor Tables.
- General Segment Descriptor Format.
- Goodbye to Segmentation.

The Next Chapter

The next chapter provides a detailed description of code segments (both Conforming and Non-Conforming), privilege checking, and Call Gates. The following topics are covered:

- Selecting the Active Code Segment.
- CS Descriptor.
- Accessing the Code Segment.
- Short / Near Jumps.
- Unconditional Far Jumps.
- Privilege Checking.
- Jumping from a Higher-to-Lesser Privileged Program.
- Direct Procedure Calls.
- Indirect Procedure Far Call Though a Call Gate.
- Automatic Stack Switch.
- Far Call From 32-bit CS to 16-bit CS.
- Far Call From 16-bit CS to 32-bit CS.
- Far Returns.

Real Mode Segment Limitations

Figure 13-1 on page 385 illustrates the contents of a segment register while operating in Real Mode; i.e., the upper 16 bits of the segment's 20-bit base address (aligned on a 16-byte address boundary) in the first megabyte of memory space. The logical processor automatically appends four bits of zero to the lower end to form the base address. As an example, if the programmer moves the value 1010h into the DS register

```
mov ax, 1010
mov ds, ax
```

this would set the start address of the data segment to 10100h.

As stated earlier in the book, when in Protected Mode the OS memory manager must be able to define a number of segment properties in addition to its base address (and this is not possible in a 16-bit register).

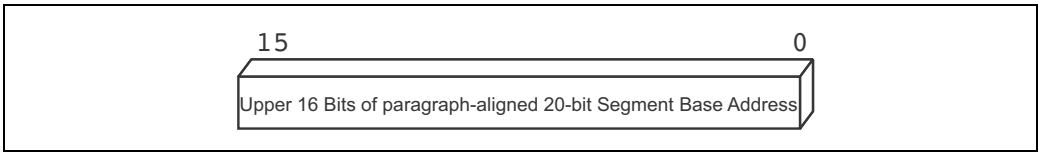
In Real Mode, a segment has the following characteristics:

- Its base address must be in the **first megabyte of memory space**. In order to have the maximum flexibility in memory allocation while operating in Protected Mode, the OS must be able to define a program's segments as residing anywhere within physical memory (above or below the 1MB address boundary).
- The **segment length is fixed at 64KB**. Unless they are incredibly small, programs and the data they manipulate virtually always occupy more than 64KB of memory space, but each segment has a fixed length of 64KB in Real

Mode. If the OS only requires a very small segment for a program's code, data or stack area, the only size available is still fixed at 64KB. This can waste memory space (albeit, not very much). If the code comprising a particular program is larger than 64KB, the programmer must set up and jump back and forth between multiple code segments. The data that a program acts upon may also occupy multiple data segments. This is a very inefficient memory organization model and one that forces the programmer to think in a very fragmented manner. It's one of the major things programmers dislike about Real Mode segmentation.

- The **segment can be read or written by any program**. In Real Mode, a segment can be accessed by any program. This is an invitation for one program to inadvertently trash another's code, data or stack area. In addition, any program can call procedures within any other program. There is no concept of restricting access to certain programs.

Figure 13-1: Segment Register Contents in Real Mode



An Important Reminder: Segment Base + Offset = Virtual Address

While this chapter (along with the two chapters that follow) provides a detailed description of segmentation, keep in mind that the address produced by adding an offset to a segment base address may *not*, in fact, be the memory address that is used to access physical memory. If the virtual-to-physical address translation mechanism (i.e., paging) is enabled, it is treated as a *virtual* (also referred to as *linear*) memory address that is subsequently submitted to the virtual-to-physical address translation logic. Upon receipt of a virtual memory address, the Paging logic uses it to perform a lookup in special address translation tables created in memory by the OS kernel. The Page Table Entry (PTE) selected by the virtual address contains the information used to translate the virtual address into a physical memory address.

The two chapters immediately following this one provide detailed discussions of code, data and stack segments, after which the subsequent chapter provides a detailed description of the address translation mechanism.

Descriptor Contains Detailed Segment Description

In a Protected Mode environment, the OS programmer must be able to specify the following characteristics of each segment:

- The base address anywhere in a 4GB virtual address range.
- Segment length (anywhere from one byte to 4GB in length).
- How the segment may be accessed:
  - A read-only data segment.
  - An execute-only code segment (contains only code and no data).
  - A code segment that also contains read-only data.
  - A read / writable data segment (contains both code and read-only data).
- The minimum privilege level a program must have in order to access the segment.
- Whether it's a code or data segment, or a special segment used only by the OS kernel and the logical processor.
- Whether the segment of information is currently present in memory or not.

In Protected Mode, it requires eight bytes (64-bits) of information to describe all of these characteristics. The Protected Mode OS memory manager must create an eight byte descriptor for each memory segment to be used by each program (including those used by the OS itself). Obviously, it would consume a great deal of processor real estate to keep the descriptors of all segments in use by all tasks on the processor chip itself. For this reason, the descriptors are stored in special tables in memory. The next section provides a description of these descriptor tables.

Segment Register—Selects Descriptor Table and Entry

When a programmer wishes to gain access to an area of memory, the respective segment register (the CS, SS, or one the data segment registers: DS, ES, FS, or GS) must be loaded with a 16-bit value that identifies the area of memory. In Real Mode, the value loaded into the segment register represents the upper 16 bits of the 20-bit start address of the segment in memory. In Protected Mode, the value loaded into a segment register is referred to as the segment selector, illustrated in the upper part (i.e., the segment register's visible part) of Figure 13-3 on page 389:

- **RPL field**. The Requester Privilege Level (RPL) field is described in "RPL Definition" on page 439 and "RPL Definition" on page 439.

14 Code, Calls and Privilege Checks

The Previous Chapter

The previous chapter introduced segment register usage in Protected Mode and the role of segment descriptors, the GDT, the LDTs, the IDT, and the general segment descriptor format. It also introduced the concept of the flat memory model. The following topics were covered:

- Real Mode Segment Limitations.
- An Important Reminder: Segment Base + Offset = Virtual Address.
- Descriptor Contains Detailed Segment Description.
- Segment Register—Selects Descriptor Table and Entry.
- The Descriptor Tables.
- General Segment Descriptor Format.
- Goodbye to Segmentation.

This Chapter

This chapter provides a detailed description of code segments (both Conforming and Non-Conforming), privilege checking, and Call Gates. The following topics are covered:

- Selecting the Active Code Segment.
- CS Descriptor.
- Accessing the Code Segment.
- Short / Near Jumps.
- Unconditional Far Jumps.
- Privilege Checking.
- Jumping from a Higher-to-Lesser Privileged Program.
- Direct Procedure Calls.

- Indirect Procedure Far Call Though a Call Gate.
- Automatic Stack Switch.
- Far Call From 32-bit CS to 16-bit CS.
- Far Call From 16-bit CS to 32-bit CS.
- Far Returns.

The Next Chapter

The next chapter provides a detailed description of Data and Stack segments (including Expand-Up and Expand-Down Stacks) and privilege checking. The following topics are covered:

- The Data Segments.
  - — General.
  - — Two-Step Permission Check.
  - — An Example.
- Selecting and Accessing a Stack Segment.
  - — Introduction.
  - — Expand-Up Stack.
  - — Expand-Down Stack.
    - – The Problem.
    - – Expand-Down Stack Description.
    - – An Example.
    - – Another Example.

Abbreviation Alert

In many cases in this chapter, the abbreviation *CS* is substituted for *code segment*.

Selecting the Active Code Segment

To execute code from a specific area of memory, the programmer must tell the logical processor what code segment the instructions are to be fetched from. This is accomplished by loading a 16-bit value (a selector) into the Code Segment (CS) register. In Real Mode, this value represents the upper 16-bits of the 20-bit zero-extended segment base address. In Protected Mode, the value loaded into the 16-bit visible portion of the CS register (see Figure 14-1 on page 418) selects an entry in either the GDT or the LDT. The selected 8-byte CS descriptor is automatically read from memory and stored in the invisible part of the CS register.

Any of the actions listed in Table 14-1 on page 417 loads a new selector into CS and causes the logical processor to begin fetching instructions from a new code segment in memory.

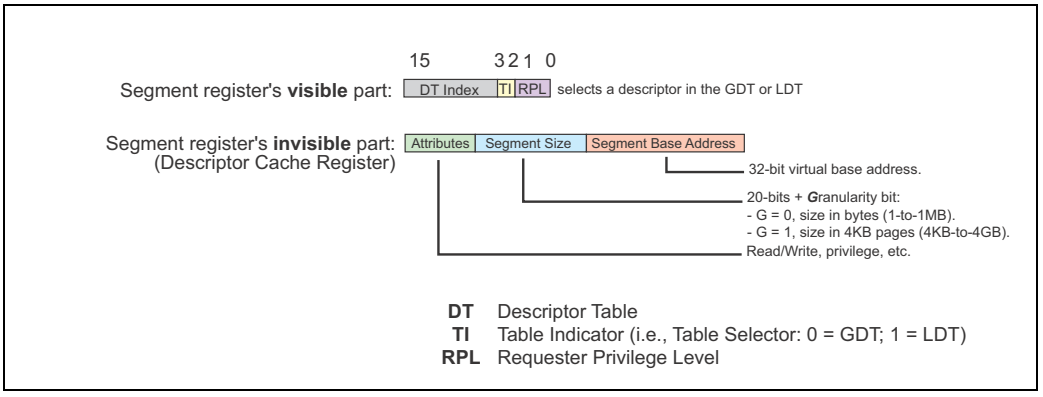
Table 14-1: Actions That Cause a Switch to a New CS

Action	Description
Execution of a **far jump** instruction	Loads the CS/Instruction Pointer register pair with new values.
Execution of a **far call** instruction	
A **hardware interrupt** or a **software exception**	In response, the logical processor loads new values into the CS/Instruction Pointer register pair from the IDT entry selected by the interrupt or exception vector.
Execution of a **software interrupt instruction**	• **INT nn**. In response, the logical processor loads new values into the CS/Instruction Pointer register pair from the IDT entry selected by the instruction's 8-bit operand. • **INT3**. In response, the logical processor loads new values into the CS/Instruction Pointer register pair from IDT entry 3. • **INTO**. In response, the logical processor loads new values into the CS/Instruction Pointer register pair from IDT entry 4. • **BOUND**. In response, the logical processor loads new values into the CS/Instruction Pointer register pair from IDT entry 5.
Initiation of a **new task or resumption of** a previously-**suspended task**	During a task switch by the x86 processor's hardware-based task switching mechanism (which most modern OSs do not use), the logical processor loads most of its registers, including CS:EIP, with values from the TSS associated with the task being started or resumed.
Execution of a **far RET** instruction	The far return address (CS and Instruction Pointer) is popped from the stack and loaded into the CS/Instruction Pointer register pair.

Table 14-1: Actions That Cause a Switch to a New CS (Continued)

Action	Description
Execution of an **Interrupt Return** instruction (IRET)	The far return address is popped from the stack and loaded into the CS / Instruction Pointer register pair.

Figure 14-1: Segment Register



CS Descriptor

CS Descriptor Selector

The value loaded into the visible part of CS (Figure 14-1 on page 418) identifies:

- The descriptor table that contains the code segment descriptor:
  - TI = 0 selects the GDT.
  - TI = 1 selects the LDT.
- The entry in the specified descriptor table. The DT (Descriptor Table) Index field selects one of 8192d entries in the selected table.
- The privilege level of the program that created the selector in the CS register. This is referred to as the Requester Privilege Level (RPL).

15 Data and Stack Segments

The Previous Chapter

The previous chapter provided a detailed description of code segments (both Conforming and Non-Conforming), privilege checking, and Call Gates. The following topics were covered:

- Selecting the Active Code Segment.
- CS Descriptor.
- Accessing the Code Segment.
- Short / Near Jumps.
- Unconditional Far Jumps.
- Privilege Checking.
- Jumping from a Higher-to-Lesser Privileged Program.
- Direct Procedure Calls.
- Indirect Procedure Far Call Though a Call Gate.
- Automatic Stack Switch.
- Far Call From 32-bit CS to 16-bit CS.
- Far Call From 16-bit CS to 32-bit CS.
- Far Returns.

This Chapter

This chapter provides a detailed description of Data and Stack segments (including Expand-Up and Expand-Down Stacks) and privilege checking when accessing data or stack segments. The following topics are covered:

- The Data Segments.
  - General.
  - Two-Step Permission Check.

  - An Example.
- Selecting and Accessing a Stack Segment.
  - Introduction.
  - Expand-Up Stack.
  - Expand-Down Stack.
    - The Problem.
    - Expand-Down Stack Description.
    - An Example.
    - Another Example.

The Next Chapter

The next chapter covers the following topics:

- Summarizes the evolution of the virtual-to-physical address translation facilities on the x86 processors and provides a backgrounder on memory and disk management.
- The concept of virtual memory is introduced as well as the advantages of address translation.
- The first and second generation virtual-to-physical address translation mechanisms are described in detail.
- The role of the Translation Lookaside Buffer (TLB) is described, as well as the Global Page feature and TLB maintenance.
- Page Directory Entries (PDEs) and Page Table Entries (PTEs) are described in detail.
- Page access permission.
- Missing page or Page Table.
- Page access history.
- 4MB pages.
- PSE-36 Mode.
- Execute Disable feature.
- Page caching rules.
- Page write protection.

A Note Regarding Stack Segments

While the stack segment is, in reality, nothing more than a read/writable data segment, it is treated separately in this chapter because it is used differently than the typical data segment.

Data Segments

General

x86 processors introduced after the 286 implement four data segment registers (as opposed to just one, DS, in the 286): DS, ES, FS and GS. They permit software to identify up to four separate data segments (in memory) that can be accessed by the currently executing program.

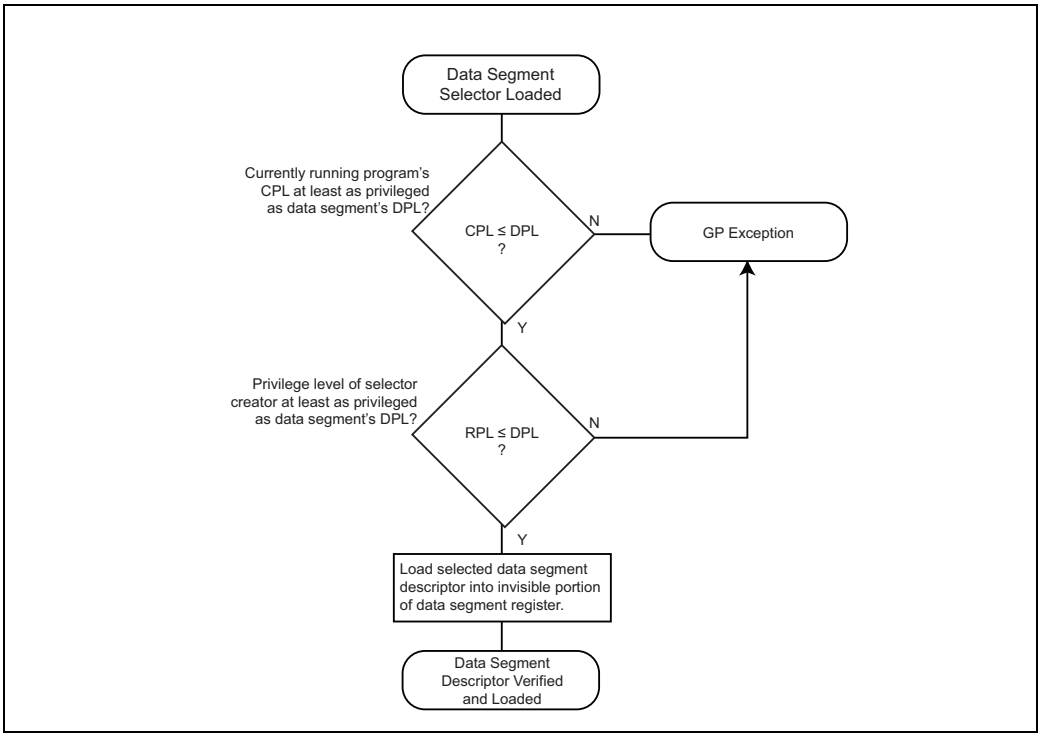
To access data within any of the four data segments, the programmer must first load a 16-bit descriptor selector into the respective data segment register. In Real Mode, the value in a data segment register specifies the upper 16-bits of the 20-bit zero-extended memory start address of the data segment. In Protected Mode, the value selects a data segment descriptor in either the GDT or LDT. Figure 15-3 on page 484 illustrates the format of a 32-bit data segment descriptor (in a 286-style 16-bit data segment descriptor, bytes 6 and 7 are reserved).

Two-Step Permission Check

In order to successfully access one or more locations in a data segment, two permission checks must be passed:

1. **Descriptor pre-load privilege check**. Refer to Figure 15-1 on page 482. The currently running program must have sufficient privilege to select the target data segment descriptor in the GDT or LDT. Assuming it does, the selected data segment descriptor is loaded into the invisible portion of the respective data segment register.
2. **Access type/limit checks**. Before any subsequent access is permitted within a data segment, the logical processor must verify that the access type is permitted (e.g., that a write is permitted) and must also verify that the specified location (i.e., offset) falls within the bounds of the targeted data segment.

Figure 15-1: Data Segment Descriptor Pre-Load Privilege Check



An Example

Consider this example (assumes code is fetched from a Non-Conforming CS with a DPL of 2):

```
mov ax, 4f36   ;load ds register
mov ds, ax     ;
mov al, [0100] ;read 1 byte from data segment into al
mov [2100], al ;write 1 byte to data segment from al
```

The value 4F36h in the DS register is interpreted by the logical processor as indicated in Figure 15-2 on page 484. The logical processor accesses LDT entry 2534 to obtain the data segment descriptor. The selector's RPL = 2 indicating that a privilege level 2 program created the selector value. Figure 15-3 on page 484

16 IA-32 Address Translation Mechanisms

The Previous Chapter

The previous chapter provided a detailed description of Data and Stack segments (including Expand-Up and Expand-Down Stacks) and privilege checking. The following topics were covered:

- The Data Segments.
  - General.
  - Two-Step Permission Check.
  - An Example.
- Selecting and Accessing a Stack Segment.
  - Introduction.
  - Expand-Up Stack.
  - Expand-Down Stack.
    - The Problem.
    - Expand-Down Stack Description.
    - An Example.
    - Another Example.

This Chapter

This chapter covers the following topics:

- Summarizes the evolution of the virtual-to-physical address translation facilities on the x86 processors and provides a backgrounder on memory and disk management.

- The concept of virtual memory is introduced as well as the advantages of address translation.
- The first and second generation virtual-to-physical address translation mechanisms are described in detail.
- The role of the Translation Lookaside Buffer (TLB) is described, as well as the Global Page feature and TLB maintenance.
- Page Directory Entries (PDEs) and Page Table Entries (PTEs) are described in detail.
- Page access permission.
- Missing page or Page Table.
- Page access history.
- 4MB pages.
- PSE-36 Mode.
- Execute Disable feature.
- Page caching rules.
- Page write protection.

The Next Chapter

The next chapter describes the operational characteristics of various types of memory targets (UC, WC, WP, WT, and WB) and the role of the Memory Type and Range Registers (MTRRs). It defines the concept of speculatively executed loads and describes issues related to the logical processor's Posted Memory Write Buffer (PMWB) and Write-Combining Buffers (WCBs).

Three Generations

Over the years, the x86 address translation mechanism has experienced three major evolutionary changes (as well as a number of smaller, incremental changes). Consequently, the author has divided the discussion into three major sections:

- **1st-generation paging**. The address translation mechanism was first introduced in the x86 product line with the advent of the 386 processor. This mechanism (including some minor enhancements added in the 486 and Pentium) is what the author refers to as the *first-generation paging* mechanism. It should be noted, however, that page address translation was actually first introduced in mainframe computers many years earlier.
- **2nd-generation paging**. The next major evolutionary jump, PAE-36 Mode, was first implemented in the Pentium Pro processor.
- **3rd-generation paging**. Part of the Intel 64 architecture.

The first and second generations are covered in this chapter. The third generation is covered in "IA-32e Address Translation" on page 983.

Demand Mode Paging Evolution

Since the advent of the 386 processor, a number of enhancements have been made to the Paging mechanism. Table 16-1 on page 495 tracks the evolutionary changes that appeared in successive generations of the x86 processor family.

Table 16-1: Paging Evolution

Processor	Enhancement	Described in
386	-	**First-generation address translation**. Virtual-to-physical address translation was first introduced to the x86 product family with the advent of the 386 processor. Using this mechanism, a 2-level lookup is used to translate a 32-bit virtual address into a 32-bit physical memory address.
486	Write Protect feature	A complete description of this minor enhancement can be found in "Example Usage: Unix Copy-on-Write Strategy" on page 569.
	Caching Rules	Minor enhancements: • CR3[PCD] and CR3[PWT] were added. A complete description can be found in "Translation Table Caching Rules" on page 585. • PCD and PWT bits were added to each Page Directory Entry (PDE). Refer to "Defining a Page's Caching Rules" on page 585. • PCD and PWT bits were added to each Page Table Entry (PTE). Refer to "Defining a Page's Caching Rules" on page 585.
Pentium	4MB Pages	The Page Size Extension (PSE) feature was added. This minor enhancement was first implemented in the Pentium and was migrated into the later versions of the 486. A complete description can be found in "4MB Pages" on page 550.
	Global pages	A minor enhancement. A complete description can be found in "Global Pages" on page 526.

Table 16-1: Paging Evolution (Continued)

Processor	Enhancement	Described in
Pentium Pro	PAE-36 Mode (2nd generation paging)	**Second-generation address translation**. Using this mechanism, a 3-level lookup is used to translate a 32-bit virtual address into a 36-bit physical memory address. A complete description can be found in "Second-Generation Paging" on page 553.
Pentium II	PSE-36 Mode	This was a minor enhancement added in the Pentium II Xeon (the very first Xeon processor). A complete description can be found in "PSE-36 Mode Background" on page 575.
	PAT feature	Page Attribute Table feature (a minor enhancement). A complete description can be found in "PAT Feature (Page Attribute Table)" on page 587.
Pentium 4	Intel 64	**Third-generation address translation**. The Intel 64 architecture introduced the third-generation address translation mechanism. Using this mechanism, a 4-level lookup can translate a 48-bit virtual address into a physical memory address up to 48-bits in width (*note*: current implementations translate a 48-bit virtual address into a 40-, 41-, or 48-bit physical address).

Background

Memory and Disk: Block-Oriented Devices

Mass storage devices are block-oriented devices. Information is stored on a disk as a series of fixed-length blocks of information and the OS manages disks in that manner. From the perspective of the OS kernel, memory is also managed as a series of fixed-length blocks—referred to as pages—of storage.

Definition of a Page

The OS kernel's memory manager (frequently referred to as the *malloc*, or memory allocation, facility) manages memory as a series of pages of information, each of a uniform size, each starting on an address boundary divisible by its

17 Memory Type Configuration

The Previous Chapter

The previous chapter covered the following topics:

- Evolution of demand mode paging on the x86 processors. Backgrounder on memory and disk management.
- Virtual memory concept and advantages of address translation.
- First and second generation virtual-to-physical address translation mechanisms.
- Role of the Translation Lookaside Buffer (TLB). Global Page feature and TLB maintenance.
- Page Directory Entries (PDEs) and Page Table Entries (PTEs).
- Page access permission.
- Missing page or Page Table.
- Page access history.
- 4MB pages.
- PSE-36 Mode.
- Execute Disable feature.
- Page caching rules.
- Page write protection.

This Chapter

This chapter describes the operational characteristics of various types of memory targets (UC, WC, WP, WT, and WB) and the role of the Memory Type and Range Registers (MTRRs). It defines the concept of speculatively executed loads and describes issues related to the logical processor's Posted Memory Write Buffer (PMWB) and Write-Combining Buffers (WCBs).

The Next Chapter

The next chapter contrasts hardware- versus software-based tasking switching and provides a conceptual overview of task switching as well as a detailed description of the hardware-based task switching mechanism. The following topics are covered:

- Hardware- vs. Software-Based Task Switching
- A Condensed Conceptual Overview
- A More Comprehensive Overview
- Hardware-Based Task Switching
  - It's Slow
  - Why Didn't OSs Use It?
  - Why Wasn't It Improved?
  - Why Does It Still Exist?
  - Introduction to the Key Elements
  - The Trigger Events
  - The Descriptors
  - The Task Register
  - TSS Data Structure Format
  - Comprehensive Task Switch Description
  - Calling Another Task
  - Task Switching and Address Translation
  - Switch from Higher-Privilege Code to Lower

Characteristics of Memory Targets

Introduction

When the logical processor must perform a memory access, it is important that it understand the operational characteristics of the target device in order to ensure proper operation. If it does not, the manner in which the memory access is accomplished may result in improper operation of the device or of the program.

Example Problem: Caching from MMIO

As an example, assume that an area of memory is populated with a series of memory-mapped IO (MMIO) registers associated with one or more devices.

Now assume that the program performs a 4-byte memory read to obtain the status of a device from its 32-bit, device-specific status register. If the logical processor were to assume that the region of memory being accessed is cacheable, it would perform a lookup in its caches and, in the event of a cache miss, would initiate a memory read to obtain not only the four requested locations, but would in fact read from all locations that encompass the line within which the desired four locations reside. This could result in a serious problem. The contents of *all* of the memory-mapped IO ports within that line of memory space would be read and cached in the processor. If the program subsequently issued a request to access any of those locations, it would result in a cache hit and:

- **If it's a read**: the requested data is supplied from the cache, *not* from the actual IO device that implements that memory-mapped IO port. This means that the data or status obtained would not represent the current, up-to-date contents of the location read. This desynchronization between a device driver and its related device can result in erroneous operation.
- **If it's a write**: the line in the cache is updated but, if the memory area is designated as WB (cacheable Write-Back) memory, the data is not written to memory. The actual memory-mapped IO device therefore does not receive the write.

Early Processors Implemented Primitive Mechanism

The example just described is but one case wherein the logical processor's lack of knowledge regarding the rules of conduct it must follow within a given memory area can result in spurious operation. In a very limited sense, the 486 and Pentium processors possessed a mechanism that permitted the OS kernel to define the characteristics of a region of memory. Each PTE (Page Table Entry) contained two bits, PCD and PWT, that permitted the OS to define a 4KB memory page as cacheable Write Through (WT), cacheable Write Back (WB), or uncacheable (UC) memory. This solution was insufficient for two reasons:

- The OS typically is not platform-specific and therefore doesn't necessarily know the characteristics of the various devices that populate memory space. The BIOS, on the other hand, *is* platform-specific but it is the OS and not the BIOS that sets up and maintains the Page Tables in memory.
- There are many different types of devices and some require different processor operation than that defined using the PTE's PCD and PWT bits (the WB, WT and UC memory types). Be advised that the later addition of the PAT feature [see "PAT Feature (Page Attribute Table)" on page 587] permitted the OS to assign any memory type to a page).

Solution/Problem: Chipset Memory Type Registers

When a program executing on the 486 or the Pentium had to initiate a memory access, the processor's internal hardware consulted the PTE[PCD] and PTE[PWT] bits to determine the rules of conduct to follow within the addressed memory page. If the memory access necessitated the performance of a transaction on the FSB (Front-Side Bus), during the memory transaction the processor transmitted the state of the PCD and PWT bits on its PCD and PWT output pins. Using the memory address output by the processor, the chipset would consult a chipset design-specific register set to determine the rules of conduct to be followed within the addressed memory area. If there was a disagreement between the OS-defined rules (as output on PCD and PWT) and the chipset's rules (as defined by the contents of its register set), the chipset would defer to the more conservative memory type (i.e., the less aggressive of the two memory types). As an example, if the processor initiated a cache line read on the FSB and the chipset said it was UC (uncacheable) memory while the processor said it was WB (cacheable Write Back) memory, the chipset would inform the processor that the entire line would *not* be returned (as the processor requested), but rather just the requested data item that caused a cache miss would be returned.

The chipset's register set was programmed by the BIOS at startup time. The problem with this approach is that the chipset's register set was implemented in a chipset-specific manner outside the scope of any industry standard specification. There would therefore have to be a separate version of the BIOS to cover all of the possible chipset types that would be used on system boards incorporating the BIOS.

Solution: Memory Type Register Set

With the advent of the Pentium Pro, Intel migrated the memory type configuration register set that had historically resided in the chipset into the processor itself. This register set is referred to as the Memory Type and Range Registers (MTRRs). While the MTRRs were, in fact, implemented identically in all members of the P6 and Pentium 4 processor families, they were *not* part of the x86 ISA specification and therefore not guaranteed to be implemented identically (or, for that matter, at all) in any given processor model. With the advent of the Pentium 4, however, the MTRRs were officially defined as part of the x86 ISA (and the register names are preceded by *IA32*). They are implemented as MSRs and are accessed using the RDMSR and WRMSR instructions.

18 Task Switching

The Previous Chapter

The previous chapter described the operational characteristics of various types of memory targets (UC, WC, WP, WT, and WB) and the role of the Memory Type and Range Registers (MTRRs). It defined the concept of speculatively executed loads and described issues related to the logical processor's Posted Memory Write Buffer (PMWB) and Write-Combining Buffers (WCBs).

This Chapter

This chapter contrasts hardware- versus software-based tasking switching and provides a conceptual overview of task switching before providing a detailed description of the hardware-based task switching mechanism. The following topics are covered:

- Hardware- vs. Software-Based Task Switching
- A Condensed Conceptual Overview
- A More Comprehensive Overview
- Hardware-Based Task Switching
  - It's Slow
  - Why Didn't OSs Use It?
  - Why Wasn't It Improved?
  - Why Does It Still Exist?
  - Introduction to the Key Elements
  - The Trigger Events
  - The Descriptors
  - The Task Register
  - TSS Data Structure Format
  - Comprehensive Task Switch Description
  - Calling Another Task
  - Task Switching and Address Translation
  - Switch from Higher-Privilege Code to Lower

The Next Chapter

The next chapter provides a detailed description of interrupt and exception handling in Protected Mode. This includes detailed coverage of:

- The IDT.
- Interrupt and Trap Gate operation.
- Task Gate operation.
- Interrupt and exception event categories.
- State save (and stack selection).
- The IRET instruction.
- Maskable hardware interrupts.
- Non-Maskable Interrupt (NMI).
- Machine Check exception.
- SM interrupt (SMI).
- Software interrupt instructions.
- Software exceptions.
- Interrupt / exception priority.

Hardware- vs. Software-Based Task Switching

The 386 introduced a number of well-received features, but for many OS vendors, its hardware-based task switching mechanism was not considered one of them. On the one hand, it permits the automation of the OS scheduler's job of switching from one task to another after the current task's timeslice has expired. On the other hand, the hardware mechanism's indulgence in an excessive validity checks renders it ponderously slow.

As a result, major OS vendors chose not to utilize the hardware mechanism and instead implemented task switching solely under the control of software (the exact implementation is OS design-specific). In tacit recognition of this reality, the hardware switching mechanism is disabled in IA-32e Mode (any attempted use of it is, in fact, considered illegal and results in an exception). The hardware mechanism is, however, supported in IA-32 Mode and is, for completeness, described in detail in this chapter. The two sections in this chapter entitled:

- "A Condensed Conceptual Overview" on page 631,
- and "A More Comprehensive Overview" on page 631

provide an introduction to task switching applicable to both the software- and -hardware-based mechanisms. A description of software-based task switching can be found in "Scheduler's Software-Based Task Switching Mechanism" on page 977.

A Condensed Conceptual Overview

The task switching concept is simple:

1. The OS scheduler selects the next task to run.
2. It initializes the logical processor's register set with the appropriate startup values.
3. It triggers a hardware timer (the timeslice timer; typically the Local APIC timer is used) configured to run for the timeslice assigned to that task (e.g., 10ms).
4. The logical processor starts executing the task and continues to do so until either:
   — The timer expires and generates an interrupt.
   — The task requires something that will take a while to complete (e.g., it issues a request to the OS to load some information from disk to memory). In this case, the OS scheduler will suspend the task (more on this later).
5. Assuming the task's timeslice has expired (i.e., the timer generates an interrupt), the event interrupts the execution of the task and returns control back to the OS kernel (specifically, to the task scheduler).
6. The scheduler suspends the task by recording the state of the logical processor's register set in a special data structure the scheduler has associated with that task.
7. It then selects the next task to start or resume and goes back to step 2.

A More Comprehensive Overview

It should be stressed that the details of task switching are OS design-specific. This discussion is conceptual (and general) in nature and applies to both software- and hardware-based task switching.

The Scheduler and the Task Queue

Some of the critical components involved in task switching are:

- **Task Scheduler**. The kernel's task scheduler is responsible for managing the task switching environment.
- **Task Queue**. Maintained by the scheduler, the **task queue** is used to keep track of:

  - The currently-running tasks.
  - Any pending events associated with those tasks. A task may have been suspended earlier after issuing a request to the OS. As an example, while a task was running, it may have issued a request to the OS for a block of information to be read from disk and placed in memory. Since this would take quite a while to complete, pending completion of the request the OS would suspend the task and start or resume another one. The scheduler would create an entry in the event queue associating the pending disk controller completion interrupt with the resumption of the previously-suspended task.
- **Timer**. The Local APIC's programmable timer is used by the task scheduler to assign the amount of time a task is permitted to execute on the logical processor before it is interrupted and control is returned to the scheduler.

Setting Up a Task

In preparation for running a task, the scheduler must:

- Load the application's startup code and data into memory. The remainder of the application remains on disk and will only be read into memory on demand (i.e., if it's required).
- Set up a series of kernel tables in memory that are necessary to support a task.
- Set up the kernel registers that tell the logical processor the location and size of these tables.

The sections that follow provide additional information about the associated tables and registers.

The Task Data Structure

Refer to Figure 18-1 on page 634. The scheduler must create a register save/restore data structure (let's call it the Task Data Structure, or TDS) in memory for each task that will be run:

- When a task is initially set up, the TDS fields will be initialized with the values to be loaded into the logical processor's register set when the task is first started.
- When starting or resuming the execution of a task, many of the logical processor's registers will be loaded from this data structure.
- When suspending a task, the registers will be saved in the task's data structure.

19 Protected Mode Interrupts and Exceptions

The Previous Chapter

The previous chapter contrasted hardware- versus software-based tasking switching and provided a conceptual overview of task switching. It then provided a detailed description of the hardware-based task switching mechanism. The following topics were covered:

- Hardware- vs. Software-Based Task Switching
- A Condensed Conceptual Overview
- A More Comprehensive Overview
- Hardware-Based Task Switching
  - It's Slow
  - Why Didn't OSs Use It?
  - Why Wasn't It Improved?
  - Why Does It Still Exist?
  - Introduction to the Key Elements
  - The Trigger Events
  - The Descriptors
  - The Task Register
  - TSS Data Structure Format
  - Comprehensive Task Switch Description
  - Calling Another Task
  - Task Switching and Address Translation
  - Switch from Higher-Privilege Code to Lower

This Chapter

This chapter provides a detailed description of interrupt and exception handling in Protected Mode. This includes detailed coverage of:

- The IDT.
- Interrupt and Trap Gate operation.

- Task Gate operation.
- Interrupt and exception event categories.
- State save (and stack selection).
- The IRET instruction.
- Maskable hardware interrupts.
- Non-Maskable Interrupt (NMI).
- Machine Check exception.
- SM interrupt (SMI).
- Software interrupt instructions.
- Software exceptions.
- Interrupt/exception priority.

A detailed description of the Local and IO APICs can be found in "The Local and IO APICs" on page 1239.

The Next Chapter

The next chapter provides a detailed description of VM86 Mode (also known as Virtual 8086 Mode). This includes the following topics:

- Switching Between Protected Mode and VM86 Mode.
- Real Mode Application's World View.
- Sensitive Instructions.
- Handling Direct IO.
- Handling Exceptions.
- Hardware Interrupt Handling in VM86 Mode
- Software Interrupt Instruction Handling
- Halt Instruction in VM86 Mode
- Protected Mode Virtual Interrupt Feature
- Registers Accessible in Real/VM86 Mode
- Instructions Usable in Real/VM86 Mode

Handler vs. ISR

The program executed to service a hardware interrupt or a software exception is commonly referred to as either a handler or an Interrupt Service Routine (ISR). For consistency and brevity's sake, the author has elected to use the term *handler.*

Real Mode Interrupt/Exception Handling

Real Mode handling of hardware and software interrupts as well as software exceptions was covered earlier in “Real Mode Interrupt/Exception Handling” on page 316. The following figures provide an overview of Real Mode event handling:

- Refer to Figure 19-1 on page 683.
- Refer to Figure 19-2 on page 684.

The remainder of this chapter focuses on interrupt and exception handling in Protected Mode.

Figure 19-1: Real Mode Interrupt Handling

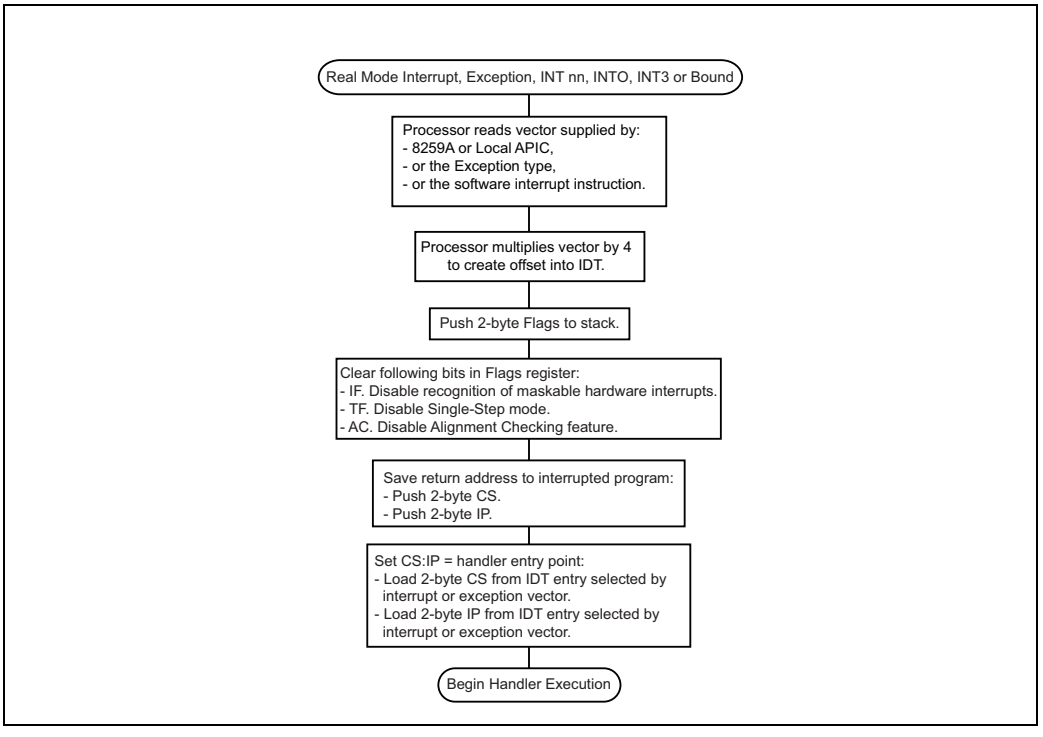
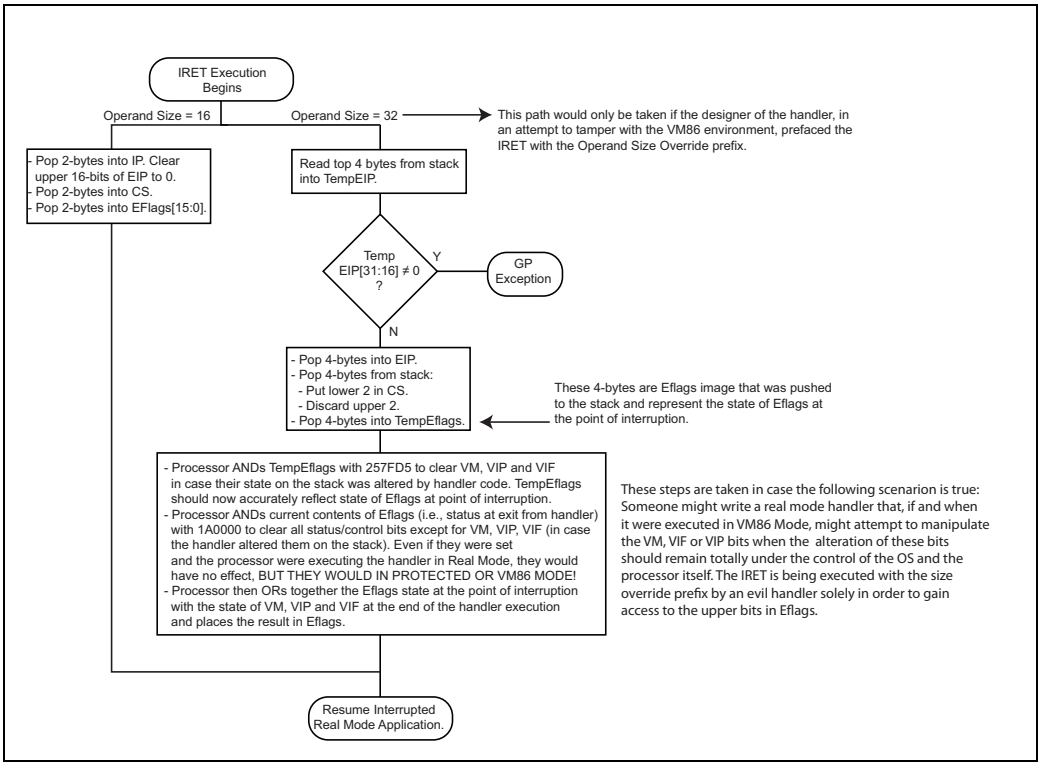


Figure 19-2: Return From Real Mode Handler To Interrupted Real Mode Application



The IDT

General

In Real Mode, the OS permits the logical processor to execute a single program at a time (i.e., multitasking is not supported). The BIOS, OS services, interrupt and exception handlers exist solely to support the program that is executing. This being the case, there is no need to restrict access to these services when the program executes a software interrupt instruction. The Protected Mode environment, on the other hand, was specifically designed to support a multitasking OS and must therefore provide protection from code being called by an entity with insufficient privilege. If the currently-executing program attempts to

20 Virtual 8086 Mode

The Previous Chapter

The previous chapter provided a detailed description of interrupt and exception handling in Protected Mode. This included detailed coverage of:

- The IDT.
- Interrupt and Trap Gate operation.
- Task Gate operation.
- Interrupt and exception event categories.
- State save (and stack selection).
- The IRET instruction.
- Maskable hardware interrupts.
- Non-Maskable Interrupt (NMI).
- Machine Check exception.
- SM interrupt (SMI).
- Software interrupt instructions.
- Software exceptions.
- Interrupt/exception priority.

This Chapter

This chapter provides a detailed description of VM86 Mode (also known as Virtual 8086 Mode). This includes the following topics:

- Switching Between Protected Mode and VM86 Mode.
- Real Mode Application's World View.
- Sensitive Instructions.
- Handling Direct IO.
- Handling Exceptions.
- Hardware Interrupt Handling in VM86 Mode
- Software Interrupt Instruction Handling
- Halt Instruction in VM86 Mode

- Protected Mode Virtual Interrupt Feature
- Registers Accessible in Real/VM86 Mode
- Instructions Usable in Real/VM86 Mode

The Next Chapter

The next chapter introduces the MMX register set and the original MMX instruction set. The SIMD programming model is introduced, how to deal with unpacked data as well as math underflows and overflows, and the elimination of conditional branches. Handling a task switch is described and the instruction set syntax is introduced.

A Special Note

The terms DOS task, VM86 task, and Real Mode task may be used interchangeably in this chapter (the vast majority of VM86 tasks are DOS tasks and, as such, intended to run in Real Mode). It should not be construed, however, that only DOS tasks are VM86 candidates. Any Real Mode application executed by a multitasking OS must be run under VM86 Mode.

Real Mode Applications Are Dangerous

The chapter entitled "Multitasking-Related Issues" on page 367 introduced some of the ways in which a Real Mode application might prove disruptive in a multitasking environment:

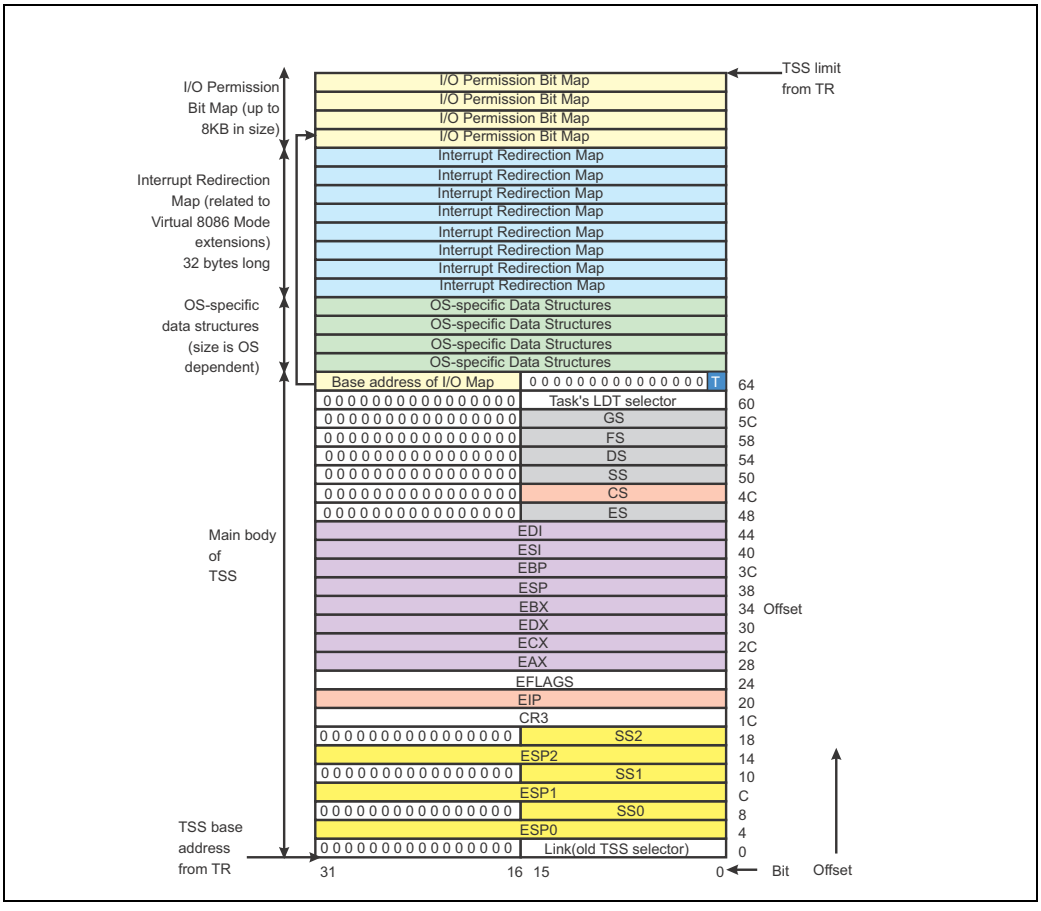
- Access memory belonging to currently-suspended programs.
- Communicate directly with IO ports (and thereby alter the state of device adapters).
- Call OS kernel code (including procedures it may not be allowed to access).
- Execute the CLI or STI instruction to disable or enable recognition of maskable hardware interrupts. The PUSHF and POPF instructions can also be used to change the state of the Eflags[IF] bit).
- Utilize a software interrupt instruction to call the BIOS or the Real Mode OS. A Real Mode application assumes it's running under a Real Mode OS rather than a multitasking, Protected Mode OS. Consequently, all OS calls initiated by the application should be intercepted and passed to the host OS (or another program that substitutes for the DOS OS).

Solution: a Watchdog

When the scheduler switches to a Real Mode application, it sets the Eflags[VM] bit to one (see Figure 20-2 on page 787). This activates a logical processor mechanism (the VM86 logic) that monitors the behavior of the application on an instruction-by-instruction basis. Any operation that might prove destabilizing to the overall multitasking environment (referred to as a *sensitive* operation) is intercepted and an exception is generated to inform Virtual Machine Monitor (VMM) handler. There are a number of elements associated with this mechanism:

1. **Monitor logic**. The VM86 hardware detects the attempted execution of the sensitive instructions. First introduced in the 386, this mechanism was improved with the Pentium's addition of the VM86 Extensions (VME) feature.
2. **GP exception**. A GP exception is generated when a sensitive operation is detected.
3. **Monitor program**. If the GP exception handler determines it was invoked by the VM86 logic, it calls a special privilege level 0 procedure referred to as the VMM (Virtual Machine Monitor) to handle the event.
4. **TSS**. The scheduler creates a TSS data structure (see Figure 20-1 on page 786) for each Real Mode application. Several TSS elements are specifically-associated with VM86 Mode:
   - IO permission bitmap.
   - Interrupt redirection bitmap.
   - The VM bit in the Eflags register field.
   - The IOPL field in the Eflags register field.
5. **VM86 Extensions**. An OS may or may not activate the VM86 Mode Extensions by setting CR4[VME] = 1. If it is enabled, the following elements come into play:
   - Eflags[VIF] and Eflags[VIP] bits.
   - Interrupt redirection bitmap consultation.
6. **IOPL threshold**. The threshold value in Eflags[IOPL].

Figure 20-1: Task State Segment (TSS)



21 The MMX Facilities

The Previous Chapter

The previous chapter provided a detailed description of VM86 Mode (also known as Virtual 8086 Mode). This included the following topics:

- Switching Between Protected Mode and VM86 Mode.
- Real Mode Application's World View.
- Sensitive Instructions.
- Handling Direct IO.
- Handling Exceptions.
- Hardware Interrupt Handling in VM86 Mode
- Software Interrupt Instruction Handling
- Halt Instruction in VM86 Mode
- Protected Mode Virtual Interrupt Feature
- Registers Accessible in Real/VM86 Mode
- Instructions Usable in Real/VM86 Mode

This Chapter

This chapter introduces the MMX register set and the original MMX instruction set. The SIMD programming model is introduced, how to deal with unpacked data as well as math underflows and overflows, and the elimination of conditional branches. Handling a task switch is described and the instruction set syntax is introduced.

The Next Chapter

The next chapter describes the SSE, SSE2 and SSE 3 instruction sets and summarizes the SSSE3, SSE4.1 and SSE4.2 instruction sets.

Introduction

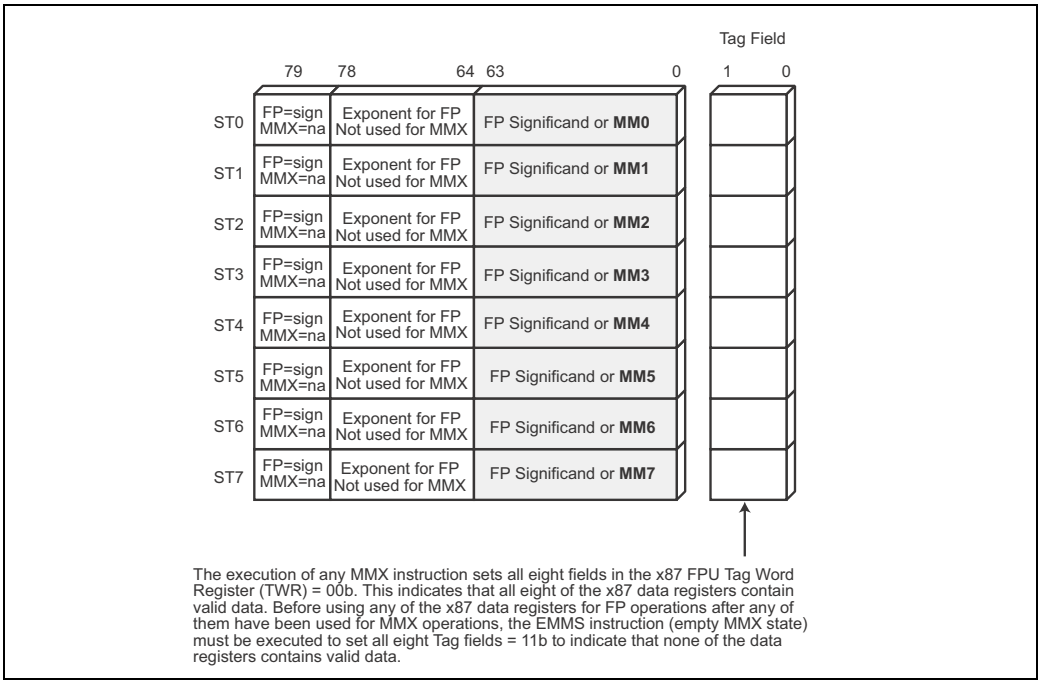
The MMX instruction set was first introduced in the P55C version of the Pentium and consisted of 47 new instructions. In addition, there are eight MMX data registers (MM0 - MM7; see Figure 21-1 on page 836). As shown in the illustration, the lower 64-bits of the x87 FPU data registers perform double-duty:

- They are used as MMX data registers when MMX code is executed.
- They are used as x87 FPU data registers when x87 FPU code is executed.

Over the years, the core concept introduced with the advent of MMX—instructions capable of simultaneously operating on multiple data items packed into wide registers—has continued to expand as Intel introduced the SSE (Streaming SIMD Extensions, where SIMD stands for ***S***ingle ***I***nstruction operating on ***M***ultiple ***D***ata items), SSE2, SSE3, SSSE3 (Supplemental SSE3), SSE4.1 and SSE4.2 instruction sets.

This chapter is not intended as an in-depth look at the initial MMX instruction set. Rather, it provides an overview of the basic concepts introduced with the advent of the MMX instruction set.

Figure 21-1: MMX Register Set



Detecting MMX Capability

Whether or not a processor supports MMX is detected by executing a CPUID request type 1 in response to which the processor capabilities bit mask is returned in the EDX register (bit 23 = 1 indicates the processor supports MMX).

The Basic Problem

Assumptions

Refer to Figure 21-2 on page 840. As an example, assume that there are two video frame buffers in memory (it should not be assumed, however, that MMX is only intended for processing video data) and that the current video mode has the following characteristics:

- Each location in the two buffers represents the color of one pixel. The first buffer location corresponds to the first pixel on the left end of the first line of pixels on the screen, the second buffer location corresponds to the second pixel on the left end of the first line of pixels on the screen, etc.
- A single location contains 8-bits (one byte), so a pixel can be any one of 256 possible colors (as represented by the values 00h - FFh).
- The video controller is currently operating at a resolution of 1024 x 786, so each of the two video frame buffers consists of 786,432 locations.

The Operation

Now assume that the programmer wants to:

1. Read the byte from the first location of one buffer,
2. Read the first location of the other buffer,
3. Add the two bytes together, and
4. Store the result back into the first location of the second buffer.

Repeat the operation for every pixel in the two frame buffers.

Example: Processing One Pixel Per Iteration

This could be accomplished in the following manner:

1. Read a byte (a pixel) from buffer one into a 1-byte register (e.g., AL).
2. Read the corresponding byte from buffer two into another 1-byte register (e.g., BL).
3. Add AL and BL together and store the result in the respective location in buffer two.
4. Since the add may result in the generation of a carry, the programmer has to decide whether to discard the carry or to factor it into the result. If the possibility of a carry must be dealt with, the programmer must include a conditional branch after the add that will either:

   — jump to the code that handles the carry,
   — or loop back to process the next pixel from the two buffers.

As indicated in the illustration, this would result in 786,432 x 2 memory reads and 786,432 memory writes. This code would generate a tremendous number of memory accesses which may or may not hit on the logical processor's internal caches. Any misses would result in memory transactions being performed on the processor's external interface. This would degrade performance in two ways:

- In a multiprocessor system wherein the processors share the same external interface, the interface bandwidth available to the other processor(s) could be substantially impacted.
- Since the external interface typically operates at a substantially slower rate of speed than the logical processor, the memory accesses would be time consuming.

Example: Processing Four Pixels Per Iteration

The number of memory accesses could be reduced by reading four bytes at a time from each buffer:

1. Read four bytes from one buffer into a 32-bit GPR register (e.g., EAX).
2. Read the corresponding four bytes from the other buffer into another 32-bit GPR register (e.g., EBX).
3. Add EAX and EBX together and store the result in one of the buffers.

There is a problem inherent in such a simplistic approach. The four bytes read

22 The SSE Facilities

The Previous Chapter

The previous chapter introduced the MMX register set and the original MMX instruction set. The SIMD programming model was introduced, how to deal with unpacked data as well as math underflows and overflows, and the elimination of conditional branches. Handling a task switch was described and the instruction set syntax was introduced.

This Chapter

This chapter describes the SSE, SSE2 and SSE 3 instruction sets and summarizes the SSSE3, SSE4.1 and SSE4.2 instruction sets. It also completes the discussion of the IA-32 programming environment.

The Next Chapter

The next chapter provides a detailed description of the IA-32e OS environment. The following topics are covered:

- Mode Switching Overview.
- Virtual Memory Addressing in IA-32e Mode.
- In 64-bit Mode, Hardware-Enforced Flat Model.
- 64-bit Instruction Pointer.
- Instruction Fetching.
- RIP-Relative Data Accesses.
- Changes To Kernel-Related Registers and Structures.
- Address Translation Mechanism.
- GDT/LDT Descriptor Changes.
- GDT and GDTR Changes.
- LDT and LDTR Changes.
- IDT/IDTR and Interrupt/Exception Changes.
- Interrupt/Trap Gate Operational Changes.

- IRET Behavior.
- IA-32e Call Gate Operation.
- TR and TSS Changes.
- Register Set Expansion (in 64-bit Mode).
- Scheduler's Software-Based Task Switching Mechanism.

Chapter Objectives

This chapter is not intended to provide a detailed description of each instruction in the SSE instruction sets. That role is already more than adequately fulfilled by the Intel and AMD instruction set reference manuals. Rather, the intention here is two-fold:

- To provide a fundamental understanding of the SSE architecture and how it works.
- To provide additional descriptions of some of the more odd or interesting instructions in the SSE instruction sets.

The SSE facilities are described in the same order in which they were introduced: SSE, SSE2, SSE3, SSSE3, SSE4.1, and SSE 4.2.

SSE: MMX on Steroids

As shown in Table 22-1 on page 852, application performance enhancement has been steadily addressed over the years by the expanding role of the SIMD programming model. It would be incorrect, however, to describe the SSE facilities solely as an expansion of MMX's SIMD programming model. As will be demonstrated in this chapter, while many of the SSE instructions do, in fact, expand on the SIMD programming model, many other non-SIMD instructions were added to address application-specific performance issues.

Table 22-1: Evolution of SIMD Model

Instruction Set	Introduced in	Description
MMX	Pentium P55C	47 new instructions. As described in "The MMX Facilities" on page 835, the SIMD concept was first introduced with the advent of MMX: • Introduction of the SIMD model. • Eight 64-bit registers (MM0:MM7) available for SIMD operations on packed data.

Table 22-1: Evolution of SIMD Model (Continued)

Instruction Set	Introduced in	Description
SSE	Pentium III	70 new instructions. The SIMD model was expanded with the introduction of the SSE (Streaming SIMD Extensions) instruction set and register set: • Eight dedicated 128-bit data registers (XMM0 - XMM7) available for SIMD operations on packed data. • The ability to perform SIMD packed and scalar FP operations on 32-bit DP FP numerical values. • MXCSR Control / Status register added to control SSE FP operations: – SSE FP exception masking and status. – Enable / disable DAZ (Denormals-As-Zero) performance enhancement mode. – Enable / disable FTZ (Flush-to-Zero) performance enhancement mode. When SSE was originally introduced, it was under the name Internet SSE (the word *Internet* was appended to just about everything during those crazy 1990s).
SSE2	130nm Pentium 4	144 new instructions. • Added the ability to perform both scalar and packed FP operations on 64-bit DP FP numbers. • The programmer can pack two, 64-bit DP FP numbers in each of two 128-bit XMM registers and then perform a packed FP operation on them (or between two numbers packed in an XMM register and two in memory). • MMX instructions enhanced to perform operations on data items packed in the XMM registers (prior to this, MMX instruction could only operate on data in MMX registers). • The CLFLUSH, MFENCE, LFENCE and new streaming store (commonly referred to as non-temporal store) instructions were added. • The PAUSE instruction was added to enhance performance when Hyper-Threading is enabled.

Table 22-1: Evolution of SIMD Model (Continued)

Instruction Set	Introduced in	Description
SSE3	90nm Pentium 4	13 new instructions. • One x87 FPU instruction (FISTTP) that improves x87 FP-to-integer conversion. • One SIMD integer instruction providing a specialized 128-bit unaligned data load. • Nine new SIMD FP instructions: – 3 instructions that enhance performance of Load / Move / Duplicate operations. – 2 instructions that perform simultaneous add / subtract operations on SP FP numbers packed into a pair of XMM registers. – 4 instructions that perform horizontal rather than vertical add and subtract operations on packed FP numbers. • Two thread-synchronization instructions (MONITOR and MWAIT) that provide a more elegant solution than the PAUSE instruction (added in SSE2) in applications employing Hyper-Threading. • They can use GPRs rather than MMX or SSE registers.

23 IA-32e OS Environment

The Previous Chapter

The previous chapter described the SSE, SSE2 and SSE 3 instruction sets and summarized the SSSE3, SSE4.1 and SSE4.2 instruction sets.

This Chapter

This chapter provides a detailed description of the IA-32e OS environment. The following topics are covered:

- Mode Switching Overview.
- Virtual Memory Addressing in IA-32e Mode.
- In 64-bit Mode, Hardware-Enforced Flat Model.
- 64-bit Instruction Pointer.
- Instruction Fetching.
- RIP-Relative Data Accesses.
- Changes To Kernel-Related Registers and Structures.
- Address Translation Mechanism.
- GDT/LDT Descriptor Changes.
- GDT and GDTR Changes.
- LDT and LDTR Changes.
- IDT/IDTR and Interrupt/Exception Changes.
- Interrupt/Trap Gate Operational Changes.
- IRET Behavior.
- IA-32e Call Gate Operation.
- TR and TSS Changes.
- Register Set Expansion (in 64-bit Mode).
- Scheduler's Software-Based Task Switching Mechanism.

The Next Chapter

The next chapter provides a detailed description of the third generation address translation mechanism utilized in IA-32e Mode.

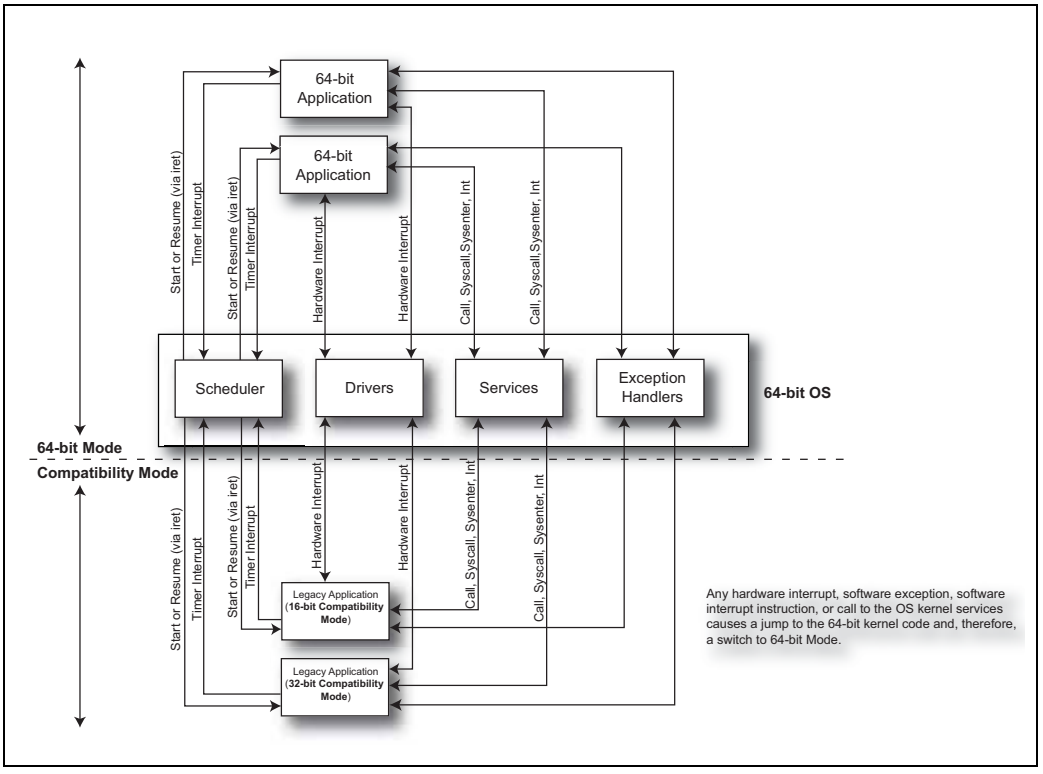
The Big Picture

Refer to Figure 23-1 on page 916. Ideally, all of the OS components are implemented as 64-bit code (i.e., they reside in 64-bit code segments and have full access to all of the logical processor's privileged, 64-bit facilities). Among other components, this would include:

- The **task scheduler**:
  1. Before starting or resuming a task, the scheduler would trigger the Local APIC timer.
  2. It then causes the logical processor to jump to the task:
     – If the task resides in a 64-bit code segment (i.e., the L bit in the CS descriptor = 1), the logical processor remains in 64-bit Mode.
     – If the task resides in a 16-bit legacy code segment (i.e., the CS descriptor's L and D bits both = 0), this causes the logical processor to automatically switch into 16-bit Compatibility Mode.
     – If the task resides in a 32-bit legacy code segment (the CS descriptor's L bit = 0 and D bit = 1), this causes the logical processor to automatically switch into 32-bit Compatibility Mode.
  3. In the background, while the logical processor executes the task, the timer continues to decrement.
  4. On timer expiration, the timer interrupt causes the logical processor to perform a far jump back to the scheduler. Since the CS descriptor selected by the far jump selects a CS descriptor wherein the L bit = 1, the logical processor switches back to 64-bit Mode (if it was in Compatibility Mode because the interrupted task was a legacy task).
- All **device drivers** (**including all hardware interrupt handlers**). In IA-32e Mode, it is a rule that all interrupt handlers must reside in 64-bit code segments:
  — Upon detection of any hardware interrupt, software exception, or the attempted execution of a software interrupt instruction (INT nn, BOUND, INT3, or INTO), the logical processor would therefore reenter 64-bit Mode (if it was in Compatibility Mode because the interrupted task was a legacy task). Note: the INTO and BOUND instructions are illegal in 64-bit Mode.

- **OS services**. OS services are typically called in one of the following ways:
  - **Far Call through a Call Gate**. The execution of a far call instruction wherein the segment selector portion of the branch target address selects a Call Gate descriptor in the GDT or LDT. Since in IA-32e Mode it is a rule that the procedure pointed to by a Call Gate must reside in a 64-bit code segment, the far call (or a far jump for that matter) causes a switch to 64-bit Mode.
  - **Software Interrupt**. The execution of a software interrupt instruction will select either an Interrupt Gate or a Trap Gate descriptor in the IDT and, since it is a rule in IA-32e Mode that all IDT descriptors must point to handlers in 64-bit code segments, the interrupt causes a switch to 64-bit Mode.
  - **SYSCALL instruction**. Used to make a call to the OS services:
    - In Intel processors, the SYSCALL instruction can only be executed successfully by 64-bit applications. Otherwise it results in an Undefined Opcode exception.
    - In AMD processors, the SYSCALL instruction can be executed in any mode.
  - **SYSENTER instruction**. Used to make a call to the OS services:
    - In Intel processors, the SYSENTER instruction can be executed in any mode.
    - In AMD processors, the SYSENTER instruction can only be executed successfully in legacy Protected Mode. Otherwise it results in an Undefined Opcode exception.
- **Software exception handlers**. In IA-32e Mode, it is a rule that all software exception handlers must reside in 64-bit code segments. Upon detection of any software exception, the logical processor would therefore reenter 64-bit Mode (if it was in Compatibility Mode because the interrupted task was a legacy task).

Figure 23-1: 64-bit OS Environment



Mode Switching Overview

Booting Into Protected Mode

The basic boot sequence is as follows:

1. Immediately after system power-up, the reset signal remains asserted until the power supply voltages have achieved their required levels and stabilized.
2. The reset signal is deasserted to the processor. One of the logical processors is selected as the Bootstrap processor, begins operation in Real Mode and initiates code fetching from the boot ROM.

24 IA-32e Address Translation

The Previous Chapter

The previous chapter provided a detailed description of the IA-32e OS environment. The following topics were covered:

- Mode Switching Overview.
- Virtual Memory Addressing in IA-32e Mode.
- In 64-bit Mode, Hardware-Enforced Flat Model.
- 64-bit Instruction Pointer.
- Instruction Fetching.
- RIP-Relative Data Accesses.
- Changes To Kernel-Related Registers and Structures.
- Address Translation Mechanism.
- GDT/LDT Descriptor Changes.
- GDT and GDTR Changes.
- LDT and LDTR Changes.
- IDT/IDTR and Interrupt/Exception Changes.
- Interrupt/Trap Gate Operational Changes.
- IRET Behavior.
- IA-32e Call Gate Operation.
- TR and TSS Changes.
- Register Set Expansion (in 64-bit Mode).
- Scheduler's Software-Based Task Switching Mechanism.

This Chapter

This chapter provides a detailed description of the third generation address translation mechanism utilized in IA-32e Mode. This includes the following topics:

- Theoretical Address Space Size.
- Limitation Imposed by Current Implementation.
- Four-Level Lookup Mechanism.
  - Address Space Partitioning.
  - The Address Translation.
    - Initializing CR3.
    - Step 1: PML4 Lookup.
    - Step 2: PDPT Lookup.
    - Step 3: Page Directory Lookup.
    - Step 4: Page Table Lookup.
  - Page Protection Mechanisms in IA-32e Mode.
    - Page Protection in Compatibility Mode.
    - Page Protection in 64-bit Mode.
    - Don't Forget the Execute Disable Feature!
- TLBs Are More Important Than Ever.
- No 4MB Page Support.

The Next Chapter

The next chapter provides a detailed description of the Compatibility SubMode of IA-32e Mode.This includes the following topics:

- Initial Entry to Compatibility Mode.
- Switching Between Compatibility Mode and 64-bit Mode.
- Differences Between IA-32 Mode and Compatibility Mode.
- Memory Addressing.
- Register Set.
- Exception and Interrupt Handling.
- OS Kernel Calls.
- IRET Changes.
- Segment Load Instructions.

Theoretical Address Space Size

Theoretically, the 3rd generation address translation mechanism utilized in IA-32e Mode would support the translation of 64-bit virtual addresses to 52-bit physical addresses. This would provide the OS with the following virtual and physical memory space sizes:

- 2^{64} virtual addressing would permit the OS to assign virtual address ranges to applications within an 16EB (exabyte) virtual address space.

- 2^{52} physical addressing would permit the OS to map a 64-bit virtual address to any physical memory address in a 4PB (petabyte) physical memory address space.

Limitation Imposed by Current Implementations

Current implementations do not support the theoretical maximum virtual or physical address ranges, however:

- A 2^{48} (256TB—terabyte) virtual address space is currently supported.
- A 2^{40} (1TB) physical memory address space (and, in some high-end AMD products, 2^{48}) is currently supported.

In other words, in IA-32e Mode the 3rd generation address translation mechanism is presented with a 48-bit virtual address (sign-extended to 64-bits to form a 64-bit canonical address) which it translates into a 40-bit (or, in some high-end AMD products, a 41- or 48-bit) physical memory address.

Four-Level Lookup Mechanism

Address Space Partitioning

Refer to Figure 24-1 on page 987. In A-32e Mode, the partitioning of the 256TB virtual address space using a 48-bit address is viewed as follows:

- The overall 48-bit 256TB virtual space is divided into 512 blocks of 512GB each. Bits 47:39 identify the target 512GB block and selects the entry in the PML4 Directory associated with the addressed 512GB virtual address block (block 66 in the illustration). PML4 Entry 4 contains the physical base address of the Page Directory Pointer Table that catalogs the location of the 512 Page Directories (PDs) associated with the targeted 512GB block.
- Each 512GB block is sub-divided into 512 blocks of 1GB each. Bits 38:30 identify the target 1GB block and selects the entry in the Page Directory Pointer Table associated with the addressed 1GB virtual address block (block 97 in the illustration). Page Directory Pointer Table Entry 97 contains the physical base address of the Page Directory that catalogs the location of the 512 Page Tables associated with the targeted 1GB block.
- Each 1GB block is sub-divided into 512 blocks of 2MB each. Bits 29:21 identify the target 2MB block and selects the entry in the Page Directory associ-

ated with the addressed 2MB virtual address block (block 8 in the illustration). Page Directory Entry 8 contains either:
  - The 4KB-aligned physical base address of the Page Table (PT) that catalogs the location of the 512 4KB pages in the targeted 2MB block;
  - Or the physical base address of the targeted 2MB page in memory.
- Each 2MB block is sub-divided into 512 pages of 4KB each. Bits 20:12 identify the target 4KB page and selects the entry in the Page Table associated with the addressed 4KB virtual page (page 34 in the illustration). Page Table Entry 34 contains the physical base address of the target 4KB page.
- The lower 12-bits (11:0) identifies the target location within the page.

25 Compatibility Mode

The Previous Chapter

The previous chapter provided a detailed description of the third generation address translation mechanism utilized in IA-32e Mode. This included the following topics:

- Theoretical Address Space Size.
- Limitation Imposed by Current Implementation.
- Four-Level Lookup Mechanism.
  - Address Space Partitioning.
  - The Address Translation.
    - Initializing CR3.
    - Step 1: PML4 Lookup.
    - Step 2: PDPT Lookup.
    - Step 3: Page Directory Lookup.
    - Step 4: Page Table Lookup.
  - Page Protection Mechanisms in IA-32e Mode.
    - Page Protection in Compatibility Mode.
    - Page Protection in 64-bit Mode.
    - Don't Forget the Execute Disable Feature!
- TLBs Are More Important Than Ever.
- No 4MB Page Support.

This Chapter

This chapter provides a detailed description of the Compatibility SubMode of IA-32e Mode. This includes the following topics:

- Initial Entry to Compatibility Mode.
- Switching Between Compatibility Mode and 64-bit Mode.
- Differences Between IA-32 Mode and Compatibility Mode.
- Memory Addressing.
- Register Set.

- Exception and Interrupt Handling.
- OS Kernel Calls.
- IRET Changes.
- Segment Load Instructions.

The Next Chapter

The next chapter provides an overview of the following:

- 64-bit Register Set.
- EFER (Extended Features Enable) Register.
- Sixteen 64-bit Control Registers.
- 64-bit Rflags Register.
- Sixteen 64-bit GPRs.
- Kernel Data Structure Registers in 64-bit Mode.
- SSE Register Set Expanded in 64-bit Mode.
- Debug Breakpoint Registers.
- Local APIC Register Set.
- x87 FPU / MMX Register Set.
- Architecturally-Defined MSRs.

Initial Entry to Compatibility Mode

This subject was introduced in “Initial Switch from IA-32 to IA-32e Mode” on page 917. A detailed description may be found in “Transitioning to IA-32e Mode” on page 1139.

Switching Between Compatibility Mode and 64-bit Mode

Once the logical processor has entered into IA-32e Mode, its mode of operation is controlled by the state of the D (Default) and L (Long Mode) bits in the current code segment’s descriptor. Additional information about switching between Compatibility Mode and 64-bit Mode may be found in the following sections:

- “CS D and L Bits Control IA-32e SubMode Selection” on page 920.
- “Scheduler’s Software-Based Task Switching Mechanism” on page 977.

Differences Between IA-32 Mode and Compatibility Mode

IA-32 Background

Except for those differences cited in this chapter, Compatibility Mode works exactly like 16- and 32-bit Protected Mode. Detailed descriptions of Protected Mode operation may be found in "Part 2: IA-32 Mode".

Unsupported IA-32 Features

The following IA-32 Mode features are not supported in IA-32e Mode (and are therefore not supported in Compatibility Mode):

- The hardware-based task switching mechanism. Due to this constraint, the following changes take effect:
  - Task Gates may not be selected in system tables (i.e., the GDT, LDTs and the IDT).
  - A far jump or a far call may not select:
    - A TSS descriptor in the GDT.
    - A Task Gate in the GDT or LDT.

    TSS descriptors are still used, however.
  - The TSS fields associated with automated task switching have been eliminated.
  - Execution of the IRET instruction when CR0[NT] = 1 does not cause a task switch.
- Real Mode.
- VM86 Mode.
- The 1st and 2nd generation address translation mechanisms.
- 4MB pages.

Changes to the OS Environment

As previously described in "IA-32e OS Environment" on page 913, the changes to the OS environment listed in Table 25-1 on page 1012 take effect in IA-32e Mode.

Table 25-1: OS Environment Changes in IA-32e Mode

Element	Description of Change
CS descriptor	A previously reserved bit, now defined as the L (Long Mode) bit, sets the logical processor's operating mode to 64-bit (L = 1) or Compatibility Mode (L = 0).
IDT	All entries in the IDT must consist of 16-byte Interrupt and Trap Gate descriptors that point to 64-bit handlers.
Call Gate	All Call Gate descriptors must be 16-byte descriptors that point to 64-bit OS services.
TSS descriptor	All TSS descriptors are 16-bytes long.
TSS	The TSS data structure format has been restructured to: • Support the IST mechanism (refer to "Interrupt / Exception Stack Switch" on page 976). • Eliminate the register save / restore area used by the automated task switch mechanism (because it is not supported). • Eliminate the Interrupt Redirection bitmap (because VM86 Mode is not supported). • Eliminate the Link field used by the automated task switch mechanism. • Eliminate the debug Trap bit.
LDT descriptor	All LDT descriptors are 16-bytes long.
Address Translation	• Only the 3rd generation translation mechanism is supported. • 4MB pages are not supported.
Virtual address	Although the virtual addresses generated by the legacy segmentation mechanism are 24- or 32-bits in length, they are zero-extended to form a 64-bit virtual address (in canonical form).
Physical address	Current implementations support a 2^{40} (1TB), 2^{41}, or 2^{48} physical address space.
CR3	CR3 is 64-bits wide enabling the top-level address translation table (the PML4) to be located anywhere in physical memory.

26 64-bit Register Overview

The Previous Chapter

The previous chapter provided a detailed description of the Compatibility Sub-Mode of IA-32e Mode. This included the following topics:

- Initial Entry to Compatibility Mode.
- Switching Between Compatibility Mode and 64-bit Mode.
- Differences Between IA-32 Mode and Compatibility Mode.
- Memory Addressing.
- Register Set.
- Exception and Interrupt Handling.
- OS Kernel Calls.
- IRET Changes.
- Segment Load Instructions.

This Chapter

This chapter provides an overview of the following:

- 64-bit Register Set.
- EFER (Extended Features Enable) Register.
- Sixteen 64-bit Control Registers.
- 64-bit Rflags Register.
- Sixteen 64-bit GPRs.
- Kernel Data Structure Registers in 64-bit Mode.
- SSE Register Set Expanded in 64-bit Mode.
- Debug Breakpoint Registers.
- Local APIC Register Set.
- x87 FPU / MMX Register Set.
- Architecturally-Defined MSRs.

The Next Chapter

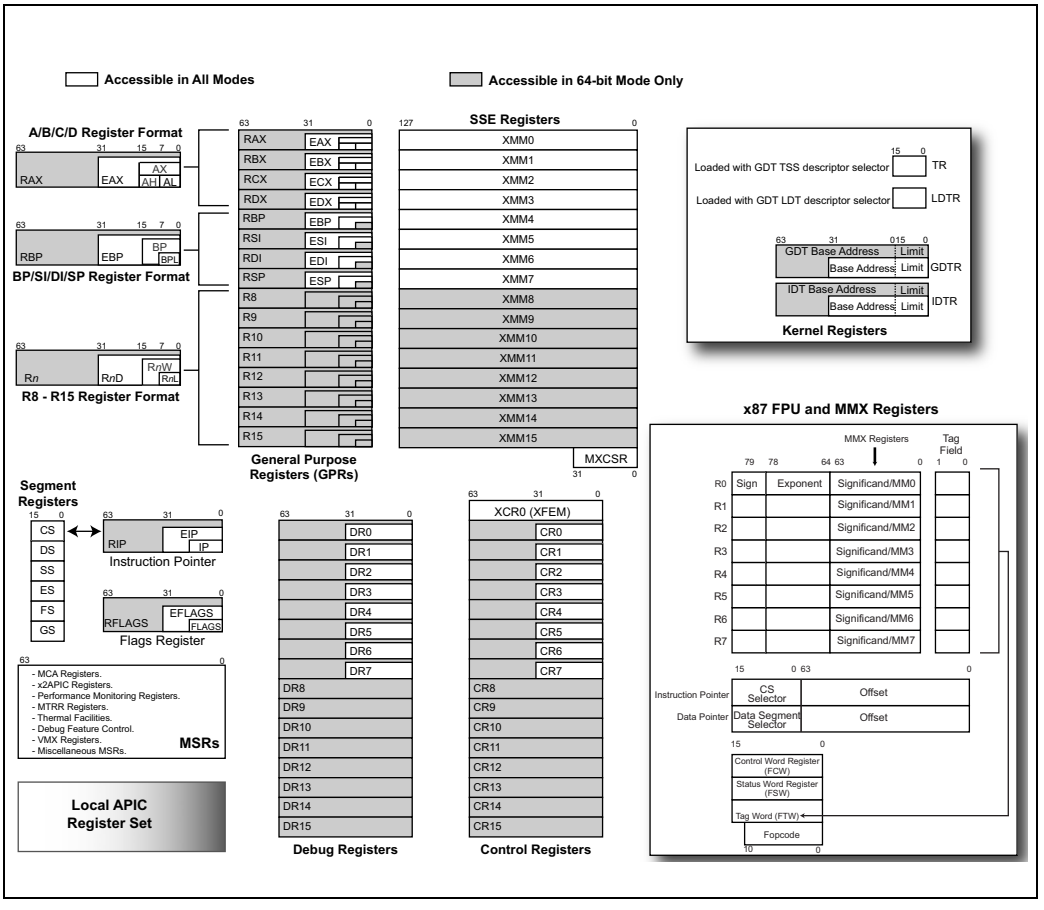
The next chapter describes the following topics:

- Switching to 64-bit Mode.
- The Defaults.
- The REX Prefix.
- Addressing Memory in 64-bit Mode.
  - 64-bit Mode Uses a Hardware-Enforced Flat Model.
  - Default Virtual Address Size (and overriding it).
  - Actual Address Size Support: Theory vs. Practice.
  - Canonical Address.
  - Memory-based Operand Address Computation.
  - RIP-relative Data Addressing.
  - Near and Far Branch Addressing.
- Immediate Data Values in 64-bit Mode.
- Displacements in 64-bit Mode.

Overview of 64-bit Register Set

Figure 26-1 on page 1025 illustrates the registers that are visible to the programmer when the logical processor is operating in the 64-bit SubMode of IA-32e Mode. A description of the registers may be found in this chapter. A description of segment register usage in 64-bit Mode can be found in "Segment Register Usage in 64-bit Mode" on page 927.

Figure 26-1: Intel 64 Register Set



EFER (Extended Features Enable) Register

The EFER register (an MSR), pictured in Figure 26-2 on page 1026, plays a central role when switching a logical processor between legacy Protected Mode and IA-32e Mode. The bit critical to the switching process is the EFER[LME] bit (Table 26-1 on page 1026 describes the register's bit assignment).

Figure 26-2: EFER (Extended Features Enable) Register

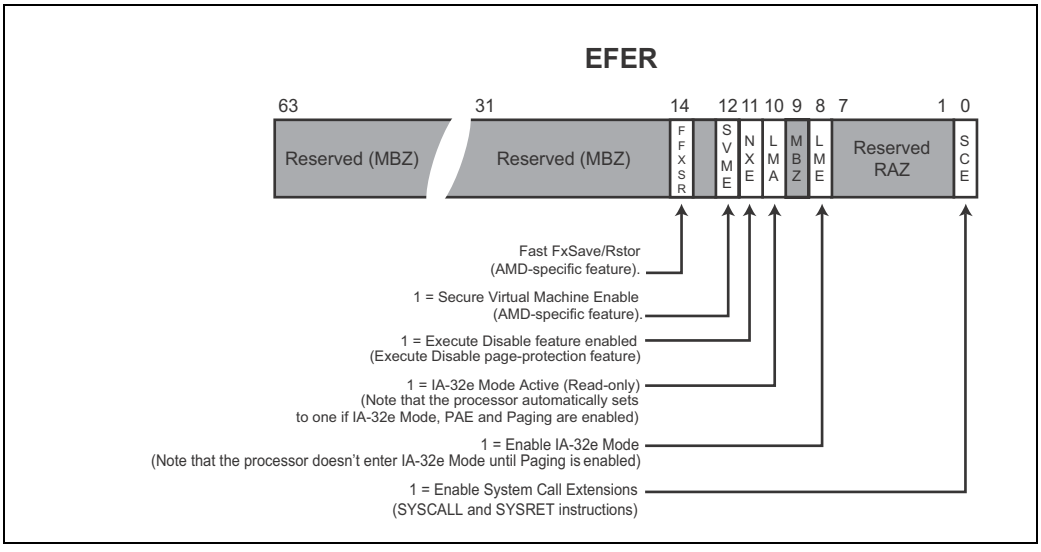


Table 26-1: EFER Register Bit Assignment

Bit	Description
SCE	**System Call Enable**. When set to one by the OS, enables the execution of the **SYSCALL** and **SYSRET** instructions which are used to make calls to the OS kernel. The OS sets this bit once it has set up the MSRs (STAR, LSTAR, CSTAR, SFMASK) used by these instructions: • Due to low overhead, these instructions provide applications a way to perform OS kernel calls very quickly. • Accomplished using predefined call / return points. The logical processor skips many of the normal type and limit checks when changing segments (CS and SS). • The call entry points and return info are defined in a set of MSRs: STAR, LSTAR, CSTAR and SFMASK. • Refer to "SysCall Instruction" on page 1018 for more information.
LME	**Enable IA-32e Mode**. The OS kernel sets this bit before paging is enabled, but the logical processor doesn't actually enter IA-32e Mode until paging is subsequently turned on with physical address extensions enabled (CR0[PG] = 1 and CR4[PAE] = 1). As an interesting side-note, Intel uses AMD's acronym (Long Mode Enable, rather than IA-32e Mode Enable) for this bit.

27 64-bit Operands and Addressing

The Previous Chapter

The previous chapter provided an overview of the following:

- 64-bit Register Set.
- EFER (Extended Features Enable) Register.
- Sixteen 64-bit Control Registers.
- 64-bit Rflags Register.
- Sixteen 64-bit GPRs.
- Kernel Data Structure Registers in 64-bit Mode.
- SSE Register Set Expanded in 64-bit Mode.
- Debug Breakpoint Registers.
- Local APIC Register Set.
- x87 FPU / MMX Register Set.
- Architecturally-Defined MSRs.

This Chapter

This chapter covers the following topics:

- Switching to 64-bit Mode.
- The Defaults.
- The REX Prefix.
- Addressing Memory in 64-bit Mode.
  - 64-bit Mode Uses a Hardware-Enforced Flat Model.
  - Default Virtual Address Size (and overriding it).
  - Actual Address Size Support: Theory vs. Practice.
  - Canonical Address.
  - Memory-based Operand Address Computation.
  - RIP-relative Data Addressing.
  - Near and Far Branch Addressing.
- Immediate Data Values in 64-bit Mode.
- Displacements in 64-bit Mode.

The Next Chapter

The next chapter describes the following 64-bit related topics:

- New Instructions.
- Enhanced Instructions.
- Invalid Instructions.
- Reassigned Instructions.
- Instructions That Default to a 64-bit Operand Size.
- Branching in 64-bit Mode.
- NOP Instruction.
- FXSAVE/FXRSTOR.
- The Nested Task Bit (Rflags[NT]).
- SMM Save Area.

Helpful Background

An understanding of the 32-bit instruction format (see "32-bit Machine Language Instruction Format" on page 155) provides the background necessary for a complete understanding of the subject matter in this chapter.

Switching to 64-bit Mode

This subject was covered earlier in "Mode Switching Overview" on page 916.

The Defaults

Unless overridden by instruction prefixes, while the logical processor is executing code from a 64-bit code segment (i.e., the L bit = 1 in the code segment descriptor), its default assumptions are set as follows:

- The default operand size = 32-bits.
- The default address size = 64-bits.

It should be noted that some instructions have a default operand size of 64-bits without the use of the REX prefix.

The REX Prefix

Problem 1: Addressing New Registers

The IA-32 instruction set's ability to specify a register as an operand is limited by the following:

- As described in "Explicit Register Specification in ModRM Byte" on page 196 (see Figure 27-1 on page 1044; please note that this figure describes the instruction format in IA-32 Mode and Compatibility Mode, *not* in 64-bit Mode), the Operand 1 (i.e., the RM field) and Operand 2 (i.e., the Reg field) fields in the ModRM byte are each three bits wide.
- As described in "Explicit Register Specification in Opcode" on page 196 (see Figure 27-2 on page 1044), the register specification field found in the primary opcode byte of some instructions is a 3-bit field.

Obviously, the constraint imposed by a 3-bit register selection field limits the selection to 1 of 8 possible registers. In 64-bit Mode, however, the programmer has the ability to specify any of 16:

- GPRs.
- XMM registers.
- Control Registers.
- Debug Registers.

This obviously requires that the Reg, RM and the primary opcode byte's register specification fields be expanded from 3- to 4-bits wide in order to address the new registers when the logical processor is in 64-bit Mode.

Refer to Figure 27-3 on page 1045. In addition:

- The Base field in the SIB (Scale / Index / Base) byte used to specify the register containing the base address of a memory-based data structure is only a 3-bit field.
- The Index field in the SIB byte used to specify the register containing the location (i.e., the index) within the data structure is only a 3-bit field.

In order to specify any of the upper eight of the sixteen GPRs as the Index and Base registers, both of these bit fields must also be expanded from 3- to 4-bits.

Figure 27-1: The ModRM Byte's Operand 1 and 2 Fields Are Each 3-bits Wide

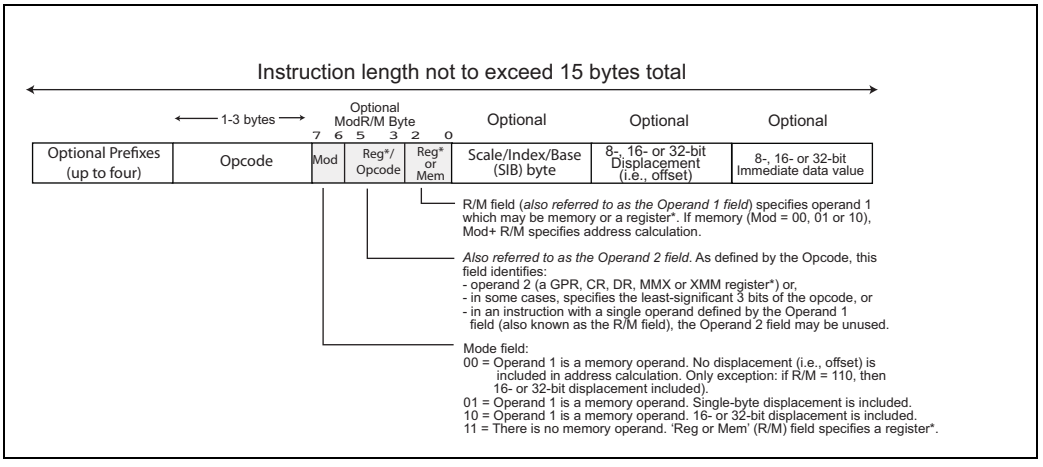
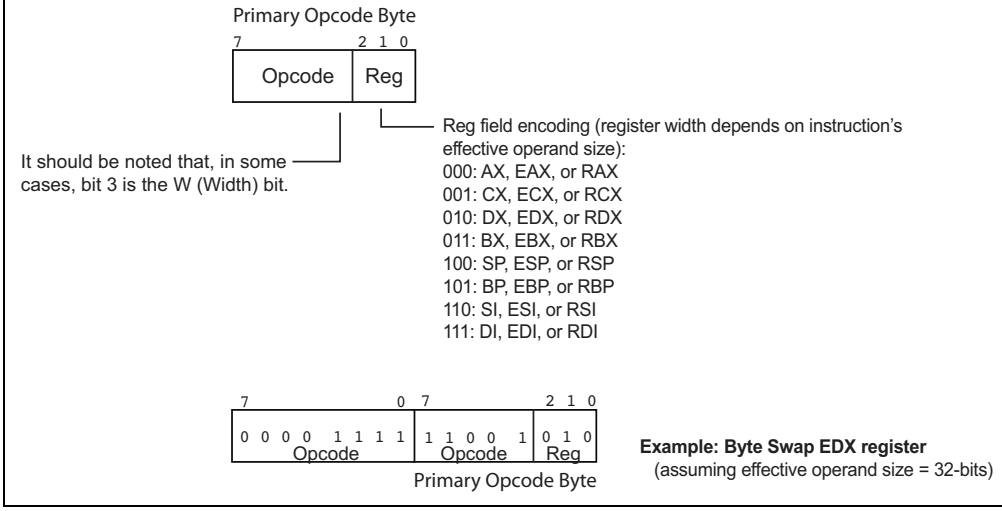


Figure 27-2: The Register Select Field in the Primary Opcode Byte Is 3-bits Wide



28 64-bit Odds and Ends

The Previous Chapter

The previous chapter described the following topics:

- Switching to 64-bit Mode.
- The Defaults.
- The REX Prefix.
- Addressing Memory in 64-bit Mode.
  - 64-bit Mode Uses a Hardware-Enforced Flat Model.
  - Default Virtual Address Size (and overriding it).
  - Actual Address Size Support: Theory vs. Practice.
  - Canonical Address.
  - Memory-based Operand Address Computation.
  - RIP-relative Data Addressing.
  - Near and Far Branch Addressing.
- Immediate Data Values in 64-bit Mode.
- Displacements in 64-bit Mode.

This Chapter

This chapter describes the following 64-bit related topics:

- New Instructions.
- Enhanced Instructions.
- Invalid Instructions.
- Reassigned Instructions.
- Instructions That Default to a 64-bit Operand Size.
- Branching in 64-bit Mode.
- NOP Instruction.
- FXSAVE/FXRSTOR.
- The Nested Task Bit (Rflags[NT]).
- SMM Save Area.

The Next Chapter

The next chapter describes the process of switching from Real Mode into Protected Mode. The following topics are covered:

- Real Mode Peculiarities That Affect the OS Boot Process.
- Typical OS Characteristics.
- Protected Mode Transition Primer.
- Example: Linux Startup.

New Instructions

General

Only two new instructions are defined when the logical processor is operating in 64-bit Mode:

- SwapGS.
- MOVSXD.

They are described in the next two sections.

SwapGS Instruction

The Problem

An application program may use the SysCall instruction to call the OS kernel services through a pre-defined entry point (the logical processor obtains the address from a special MSR register initialized by the OS). There is a problem, however. The kernel services must assume that the caller (i.e., the application program) expects the contents of the GPR registers to be preserved upon return from the kernel call. This being the case and considering that the kernel services will have to make use of one or more GPR registers in order to service the request, it will have to push the contents of one or more of the GPR registers to the stack. The problem lies in the following:

- With other system call mechanisms like Call Gates and software interrupts (INT nn), the switch to a new stack occurs automatically because there is a privilege level change occurring (unless the caller is a privilege level 0

entity). With SYSCALL, there is no automatic stack switch, so, without an instruction like this there is nowhere the called service can reliably save state information.

The SwapGS Solution

The SwapGS instruction may be used to solve this problem as shown in the following example. On entry to the kernel services, the following conditions are assumed to be true:

- The GS_Base MSR contains the 64-bit virtual base address of the caller's data area.
- The Kernel_GS_Base MSR contains the 64-bit virtual base address of a data structure within which the OS kernel stores critical information (e.g., a pointer to an empty stack area reserved for the kernel's use).

```
KernelServicesEntryPoint:
  SwapGS           ;swap KernelGSBase MSR and GSBase MSR
  mov gs:[SavedUserRSP], rsp;save caller's stack pointer
  mov rsp, gs:[KernelStackPtr]  ;set RSP = kernel stack ptr
  push rax      ;save caller's GPR(s) to kernel stack
  .
  .             ;perform requested service
  .
  pop ---       ;restore caller's GPR(s) and stack pointer
  mov rsp, gs:[SavedUserRSP];restore caller's stack pointer
  SwapGS          ; restore caller's GSBase and KernelGSBase
  ret
```

Although the stack problem doesn't exist for interrupt and exception handlers (see "Interrupt/Exception Stack Switch" on page 976), the SwapGS instruction can be used to quickly set the GS base address to point to the base address of a kernel-specific data structure that may contain information useful to the handler.

SwapGS has the following characteristics:

- SwapGS is a serializing instruction (see "Synchronizing Events" on page 618).
- The base address of the kernel-specific data structure is written to the KernelGSBase MSR (at MSR address C000_0102h) using the WRMSR instruction:
  - WRMSR may only be executed by privilege level 0 software.
  - The write will result in a GP exception if the address written to the register is not in canonical form.

- SwapGS uses a previously unused (and illegal) ModRM value accompanying the 2-byte opcode 0F01h (INVLPG; Invalidate Page Table Entry). Previously, only the memory forms (i.e., where the Mod field does not equal 11b) of this opcode were legal and the register forms (where the Mod field = 11b) were illegal. In 64-bit mode, when an 2-byte opcode of 0F01h is detected accompanied by a ModRM byte of 11 111 xxxb, the logical processor treats the xxxb bit field (i.e., the RM field) as an extension to the opcode which selects 1 of 8 instructions in a group of eight (see "Micro-Maps Associated with 2-byte Opcodes" on page 183). Currently, only RM = 000b is defined (as the SwapGS instruction) and the other seven values are currently undefined (and may be used to encode additional instructions in the future).

MOVSXD Instruction: Stretch It Out

In IA-32 and Compatibility Mode, the MOVSX instruction—Move and Sign-Extend—sign-extends a byte or word operand to a full 32-bit dword. In 64-bit Mode, the ARPL (Adjust RPL field of segment selector) is reassigned as a new instruction, MOVSXD (Move Dword and Sign Extend to 64-bits). When used with the REX prefix, it sign-extends a 32-bit value to a full 64-bits.

Enhanced Instructions

Table 28-1 on page 1078 lists instructions that support a 64-bit operand size when prefaced by the REX prefix with REX[W] = 1.

Table 28-1: Instructions Enhanced in 64-bit Mode With REX[W] = 1

Mnemonic	Opcode (hex)	Description
CDQE	98	RAX = sign-extended EAX.
CMPSQ	A7	String compare operation. Compares quadword at address RSI with quadword at address RDI and sets the Rflags status flags accordingly.

29 Transitioning to Protected Mode

The Previous Chapter

The previous chapter described the following 64-bit related topics:

- New Instructions.
- Enhanced Instructions.
- Invalid Instructions.
- Reassigned Instructions.
- Instructions That Default to a 64-bit Operand Size.
- Branching in 64-bit Mode.
- NOP Instruction.
- FXSAVE/FXRSTOR.
- The Nested Task Bit (Rflags[NT]).
- SMM Save Area.

This Chapter

This chapter describes the process of switching from Real Mode into Protected Mode. The following topics are covered:

- Real Mode Peculiarities That Affect the OS Boot Process.
- Typical OS Characteristics.
- Protected Mode Transition Primer.
- Example: Linux Startup.

The Next Chapter

The next chapter describes the process of switching from Protected Mode into the Compatibility SubMode of IA-32e Mode. It then describes making the switch from Compatibility Mode into 64-bit Mode.

Real Mode Peculiarities That Affect the OS Boot Process

Immediately after the removal of reset, an x86 processor comes up in Real Mode and, to a goodly degree, emulates the 8086. Some of the logical processor's operational characteristics when operating in Real Mode are listed below:

- **It's a 16-bit world (unless overridden)**. Unless overriden by prefacing an instruction with the Address Size and/or Operand Size Override prefixes (67h and 66h, respectively), the default segment offset address size and the default data operand size are both 16-bits.
- **An addressing anomaly**:
  - **Segment wrap-around** (see "Accessing Extended Memory in Real Mode" on page 307). When executing code on an 8086, specifying a segment base address of FFFF0h and any offset address between 0010h and FFFFh resulted in segment wraparound to the bottom of memory (specifically, to locations 00000h - 0FFEFh). The 8086 only had twenty address lines (19:0) and was therefore incapable of addressing memory above the 1MB address boundary.
  - **The HMA**. Every x86 processor since the advent of the 286, however, is capable of addressing memory above the 1MB boundary (referred to as extended memory). When operating in Real Mode, adding any offset value between 0010h - FFFFh to a segment base address of FFFF0h generates a carry bit on address bit 20 permitting software to address extended memory locations 100000h - 10FFEFh (the area of memory referred to as the HMA, or High Memory Area) without switching the logical processor into Protected Mode.
  - **A20 Gate**. Refer to "Accessing Extended Memory in Real Mode" on page 307. In order to boot the OS kernel into memory (the kernel will consume a large amount of system RAM), the logical processor must have to have the ability to address memory above the 1MB address boundary. If the A20 Gate is disabled, however, the A20 address line will always be 0 and, as a result, the logical processor will be unable to correctly address extended memory (i.e., memory above the 1MB address boundary).

Example OS Characteristics

This discussion makes the following assumptions:

- In preparation for booting the OS kernel into memory, the logical processor will be transitioned from Real Mode to Protected Mode. This is necessary

because today's highly-complex kernels are very large and will not fit in the 1MB of memory addressable in Real Mode.

- The Protected Mode memory model utilized will be a Flat Memory Model (see "IA-32 Flat Memory Model" on page 409) using the first generation virtual-to-physical address translation mechanism (i.e., paging).
- Virtually all of today's modern OSs utilize software-based task switching rather than the x86 processor's hardware-based tasking switching mechanism.
- The OS kernel and device drivers will execute at privilege level 0 while application programs will execute at level 3.

Flat Model With Paging

The discussion that follows assumes we will boot an OS that uses the Flat Memory Model and the virtual-to-physical address translation mechanism. This means that in the course of switching from Real Mode to Protected Mode, we will have to set up an appropriately formatted GDT as well as a set of virtual-to-physical address translation tables.

Software-Based Task Switching

Since the OS will not utilize the x86 hardware-based task switching mechanism, neither the GDT, the LDT (if the OS uses LDTs), nor the IDT will utilize Task Gate descriptors.

To give it full access to all of the logical processor's facilities, the OS code will execute at privilege level 0 while application programs will run at level 3. Since the logical processor is incapable of executing a jump or a call from a more-privileged to a less-privileged program, the OS task scheduler (which is privilege level 0 code) will have to use the software-based task switching mechanism described in "Scheduler's Software-Based Task Switching Mechanism" on page 977 to launch or resume an application program.

In order to avoid possible stack overflows, the OS kernel will utilize the logical processor's automatic stack switching ability (see "Automatic Stack Switch" on page 462) when making calls from one level to another. This being the case, we will have to define a TSS data structure (most modern OSs use one TSS for *all* tasks—see "Real World TSS Usage" on page 968; the TSS contains the pointers to the level 2, 1, and 0 stacks preallocated by the OS) and the TR (Task Register) is loaded with the GDT entry selector for the TSS descriptor.

Protected Mode Transition Primer

GDT Must Be In Place Before Switch to Protected Mode

At a minimum, at least a rudimentary GDT must be created in memory (see Figure 29-1 on page 1117) prior to switching to Protected Mode. The location of this initial GDT is dictated by the current state of the A20 gate (see "Accessing Extended Memory in Real Mode" on page 307):

- If the A20 Gate has not been enabled by software yet, the logical processor cannot access memory above the first MB while in Real Mode. The initial GDT must therefore be created in the first MB of memory space.
- If, on the other hand, the A20 Gate has been enabled by software, the logical processor is not solely restricted to the first MB of memory space but can also access extended memory locations 00100000h - 0010FFEFh (the HMA). In this case, the GDT may be placed in the HMA.

Keeping in mind that the desired OS memory configuration is the Flat Memory Model, the structure of the initial minimalist GDT is as follows (see Figure 29-1 on page 1117):

- **GDT size**: 24 bytes (8 bytes / entry x 3 entries).
- **Entry 0**: Entry 0 must be a null descriptor consisting of all zeros.
- **Entry 1 = CS Descriptor**. Refer to Figure 29-2 on page 1118. Entry 1 will be a code segment descriptor with the following characteristics:
  - Segment base address: 00000000h.
  - Segment size: 4GB.
  - Segment DPL: 0.
  - Other characteristics: Present bit = 1, S bit = 1, C bit = 0 for a Non-Conforming code segment, and R bit = 1 defining the CS as accessible for both code fetches and data reads.
- **Entry 2 = DS Descriptor**. Refer to Figure 29-3 on page 1119. Entry 2 will be a data segment descriptor with the following characteristics:
  - Segment base address: 00000000h.
  - Segment size: 4GB.
  - Segment DPL: 0.
  - Other characteristics: Present bit = 1, S bit = 1, R / W bit = 1 indicating it is a read / writable data segment, E bit = 0 indicating it can be used as an expand-up stack.

30 Transitioning to IA-32e Mode

The Previous Chapter

The previous chapter described the process of switching from Real Mode into Protected Mode. The following topics were covered:

- Real Mode Peculiarities That Affect the OS Boot Process.
- Typical OS Characteristics.
- Protected Mode Transition Primer.
- Example: Linux Startup.

This Chapter

This chapter describes the process of switching from Protected Mode into the Compatibility SubMode of IA-32e Mode. It then describes making the switch from Compatibility Mode into 64-bit Mode.

The Next Chapter

The next chapter provides a basic introduction to Virtualization Technology and covers the following topics:

- OS: I Am the God of All Things!
- Virtualization Supervisor: Sure You Are (:<)
- Root versus Non-Root Mode.
- Detecting VMX Capability.
- Entering / Exiting VMX Mode.
- Entering VMX Mode.
- Exiting VMX Mode.
- Virtualization Elements / Terminology.
- Introduction to the VT Instructions.

- Introduction to the VMCS Data Structure.
- Preparing to Launch a Guest OS.
- Launching a Guest OS.
- Guest OS Suspension.
- Resuming a Guest OS.
- Some Warnings Regarding VMCS Accesses.

No Need to Linger in Protected Mode

This chapter assumes that software will take the most efficient route possible from the removal of reset through Real Mode, Protected Mode, Compatibility Mode and, finally, to 64-bit Mode.

Entering Compatibility Mode

IA-32e Mode can only be entered by transitioning from legacy Protected Mode to Compatibility Mode. This transition is accomplished as follows:

1. Switch from Real Mode to legacy Protected Mode (this can be achieved using either 16- or 32-bit Protected Mode code). This topic was covered in "Transitioning to Protected Mode" on page 1113.
   — Note that there is no requirement that the address translation mechanism must be activated upon entering Protected Mode. Rather, the programmer may choose to immediately set up the 3rd generation address translation tables in preparation for the switch to the Compatibility SubMode of IA-32e Mode.

   Note: This discussion assumes the logical processor is now fetching code from a 32- rather than a 16-bit code segment. The default operand and address sizes are therefore 32-bits.

   DISABLE INTERRUPTS IN PREPARATION FOR SWITCH TO COMPATIBILITY MODE
2. Disable interrupts in preparation for switch from Protected Mode to IA-32e Mode:
   — Execute CLI to disable recognition of maskable hardware interrupts.
   — The programmer must ensure that the platform's ability to deliver an NMI has been disabled. In a PC-compatible environment, this is accomplished by executing the following:
     – mov al, 80
     – out 70,al ;performing an IO write to port 70h with bit 7 set to 1 will mask the platform's ability to deliver an NMI to the logical processor.
   — The programmer also must ensure that no instructions generate software exceptions during the switch.

SET UP IA-32E COMPLIANT DATA STRUCTURES

3. Set up the 3rd generation address translation tables (see "IA-32e Address Translation" on page 983).
4. Point CR3 (see Figure 16-19 on page 528) to the top-level address translation table (i.e., the PML4 directory). Since CR3 is only 32-bits wide in Protected Mode, the PML4 directory's physical base address must be in the lower 4GB.
5. Create an IA-32e compliant IDT containing 16-byte Interrupt Gates and Trap Gates (and no Task Gates).
6. Create an IA-32e compliant GDT containing (in addition to Protected-/Compatibility Mode-compliant data and stack segment descriptors):
   — An IA-32e compliant TSS descriptor (see Figure 23-9 on page 947).
   — An IA-32e compliant LDT descriptor (see Figure 23-8 on page 946).
   — IA-32e compliant Call Gate descriptors (see Figure 23-7 on page 945).
   — A 64-bit, privilege level 0 Non-Conforming code segment descriptor (see Figure 23-3 on page 922).
7. Create an IA-32e compliant LDT containing (in addition to Protected-/Compatibility Mode-compliant data and stack segment descriptors) IA-32e compliant Call Gate descriptors (see Figure 23-7 on page 945).
8. Create an IA-32e compliant TSS data structure (see Figure 23-9 on page 947).

EXECUTE 3-STEP PROCESS TO ENABLE IA-32E MODE

9. Enable 2nd generation address translation by setting CR4[PAE] to 1. This is the first required precondition for the transition to IA-32e Mode. Note that although the PAE feature is now enabled, address translation itself has not yet been activated (CR0[PG] is still 0).
10. Set EFER[LME] = 1 to enable IA-32e Mode. This is the second required pre-condition for the transition to IA-32e Mode. IA-32e Mode is not yet active, however.
11. Set CR0[PG] = 1 to activate paging. This is the third and final precondition. Since all three preconditions for IA-32e Mode activation have now been met, **IA-32e Mode is now activated**. The three prerequisites are:
    — CR4[PAE] = 1.
    — EFER[LME] = 1.
    — CR0[PG] = 1.
12. The L bit in the currently-active CS descriptor = 0, so the logical processor is not in 64-bit Mode. Rather, based on the state of the D bit in the selected CS descriptor, it is now in either the 16- or 32-bit Compatibility SubMode of IA-32e Mode:
    — D = 0. The logical processor is in 16-bit Compatibility Mode.
    — D = 1. The logical processor is in 32-bit Compatibility Mode.

A NOTE REGARDING IDENTITY ADDRESS MAPPING

Up until this moment, address translation was disabled. The memory address that the MOV CR0 instruction was fetched from was therefore treated as a physical rather than a virtual address. Address translation is now enabled, however, so the memory address used to fetch the next instruction (i.e., the one immediately following the MOV CR0 instruction which activated address translation) is treated as a virtual address and is therefore translated into a physical address. In order to fetch the instruction that immediately follows the MOV CR0 in physical memory, the address translation tables must translate this virtual address into the identical physical memory address (virtual = physical; referred to as identity address mapping).

AFTER SWITCH TO IA-32E MODE, LOAD SYSTEM REGISTERS

13. Execute the LIDT instruction to load the IDTR with the 16-bit size and 32-bit base address of the IA-32e compliant IDT.
14. Execute the LGDT instruction to load the GDTR with the 16-bit size and 32-bit base address of the IA-32e compliant GDT.
15. Execute the LLDT instruction to load the LDTR with the 16-bit GDT selector that points to the IA-32e compliant LDT descriptor in the GDT. In response, the logical processor loads the LDT descriptor into the invisible portion of the LDTR.
16. Execute the LTR instruction to load the TR with the 16-bit GDT selector that points to the 16-byte IA-32e compliant TSS descriptor in the GDT. In response, the logical processor loads the TSS descriptor into the invisible portion of the TR.

Switch to 64-bit Mode

1. Execute a far jump wherein the CS selector portion of the branch target address selects a 64-bit privilege level 0 Non-Conforming code segment descriptor in the GDT:
   - When the logical processor loads the CS descriptor into the invisible portion of the CS register, the one in the descriptor's L bit switches it into 64-bit Mode.
   - Segmentation is now disabled and the flat memory model is hardware-enforced: all segments (with the exception of the FS and GS data segments) have an assumed base address of 0 and a length of 2^{64} locations.
   - Since the far jump is executed while the logical processor is still in 32-bit Compatibility Mode, the offset portion of the branch target address is only 32-bits wide. The 64-bit RIP register is therefore loaded with the 32-bit address 0-extended to 64-bits. The target 64-bit code entry point must therefore reside in the lower 4GB of the 64-bit code segment.

31 Introduction to Virtualization Technology

The Previous Chapter

The previous chapter described the process of switching from Protected Mode into the Compatibility SubMode of IA-32e Mode. It then described making the switch from Compatibility Mode into 64-bit Mode.

This Chapter

This chapter provides a basic introduction to Virtualization Technology and covers the following topics:

- OS: I Am the God of All Things!
- Virtualization Supervisor: Sure You Are (:<)
- Root versus Non-Root Mode.
- Detecting VMX Capability.
- Entering / Exiting VMX Mode.
- Entering VMX Mode.
- Exiting VMX Mode.
- Virtualization Elements / Terminology.
- Introduction to the VT Instructions.
- Introduction to the VMCS Data Structure.
- Preparing to Launch a Guest OS.
- Launching a Guest OS.
- Guest OS Suspension.
- Resuming a Guest OS.
- Some Warnings Regarding VMCS Accesses.

The Next Chapter

The next chapter provides a detailed description of System Management Mode (SMM). It includes the following topics:

- What Falls Under the Heading of System Management?
- The Genesis of SMM
- SMM Has Its Own Private Memory Space
- The Basic Elements of SMM
- A Very Simple Example Scenario
- How the Processor Knows the SM Memory Start Address
- Normal Operation, (Including Paging) Is Disabled
- The Organization of SM RAM
- Entering SMM
- Exiting SMM
- Caching from SM Memory
- Setting Up the SMI Handler in SM Memory
- Relocating the SM RAM Base Address
- SMM in an MP System
- SM Mode and Virtualization

Just an Introduction?

Yes, rather than a detailed description of every aspect of virtualization, this chapter provides an introduction. Complete coverage of all aspects of virtualization would entail the addition of several hundred additional pages to an already oversize book. As such, it warrants treatment as a separate topic.

Detailed Coverage of Virtualization

Comprehensive coverage of all aspects of virtualization is available in the following MindShare class offerings:

- Comprehensive PC Virtualization:
  - — Instructor-led class. Duration: 4 days.
  - — Instructor-led internet class. Duration: 5 days.
- Fundamentals of PC Virtualization:
  - — Instructor-led class. Duration: 1 day.
  - — Instructor-led internet class. Duration: 1 day.
- Introduction to Virtualization Technology:
  - — Self-paced E-Learning Module.

- Introduction to PCI Express IO Virtualization:
  - Self-paced E-Learning Module.
- Comprehensive IO Virtualization:
  - Instructor-led class. Duration: 2 days.
  - Instructor-led internet class. Duration: 3 days.

Detailed information about MindShare's training classes and E-Learning modules may be found at www.mindshare.com.

The Intel Model

Although the basic concepts are the same, Intel and AMD have implemented vendor-specific approaches to virtualization. This chapter focuses on the Intel model.

OS: *I Am the God of All Things!*

A traditional OS (e.g., Windows XP, Windows 7, Mac OS X) has complete control of all of the logical processor's facilities:

- It executes at privilege level 0 and can therefore:
  - Access any register.
  - Control the logical processor's operational mode (i.e., whether it is in Real Mode, Protected Mode, IA-32e Mode, etc.).
  - Execute any instruction in the instruction set.
- It manages memory for all software (including itself).
- Under software control, it handles task switching among the various program's that are currently being executed.
- **Permission violation**. It manages all of the x86 protection mechanisms. If an application program attempts to touch something beyond its permission level (e.g., memory, an IO port, a Control Register, etc.), a software exception is generated which immediately returns control back to the OS kernel.
- **Action evaluation**. The kernel then evaluates the attempted action and determines how to handle it:
  - **Action permitted**. If the action would not prove detrimental to any other currently-suspended software entity, the OS may decide to permit it. In that case, the OS can execute the offending instruction itself (unlike the interrupted application program, it has sufficient privilege to do so) and then return to the interrupted program at the instruction immediately after the one that caused the exception.

— **Forbidden action**. If the attempted action is one that *would* prove detrimental to other currently-suspended software entities, the OS can handle it in either of two ways:
  – **Emulate attempted action**. The OS might choose to achieve the same goal by performing a set of actions that will not result in chaos for one or more other software entities that are currently-suspended.
  – **Abort the application**. If, in the OS's opinion, the attempted action cannot safely be permitted, it may choose to issue an alert message to the end user and then abort the errant application.

In a nutshell, the OS believes itself to be lord of all it surveys. While this *is* true under ordinary circumstances, it's *not* so when virtualization is enabled.

Virtualization Supervisor: *Sure You Are (:<)*

When the logical processor's Virtualization Technology (VT) feature (referred to as the *Virtual Mode Extensions, or VMX*) is enabled, the old gods (i.e., OSs) are subjugated to a new, all powerful God—the *Virtual Machine Monitor, or VMM* (otherwise referred to as the *hypervisor)*. The hypervisor permits guest OSs to run under its guidance, allowing each to run either for a preallocated period of time (a *timeslice*) or until the guest OS attempts a sensitive operation that might prove harmful to another, currently-suspended guest OS or to the hypervisor itself (i.e., a *sensitive* operation).

Root versus Non-Root Mode

Refer to Figure 31-1 on page 1151. When the logical processor is executing the hypervisor code, it is said to be in VMX *Root Mode*. Conversely, it is in *VMX Non-Root Mode* when it is executing one of the guest OSs.

32 System Management Mode (SMM)

The Previous Chapter

The previous chapter provided a basic introduction to Virtualization Technology and covered the following topics:

- OS: I Am the God of All Things!
- Virtualization Supervisor: Sure You Are (:<)
- Root versus Non-Root Mode.
- Detecting VMX Capability.
- Entering/Exiting VMX Mode.
- Entering VMX Mode.
- Exiting VMX Mode.
- Virtualization Elements/Terminology.
- Introduction to the VT Instructions.
- Introduction to the VMCS Data Structure.
- Preparing to Launch a Guest OS.
- Launching a Guest OS.
- Guest OS Suspension.
- Resuming a Guest OS.
- Some Warnings Regarding VMCS Accesses.

This Chapter

This chapter provides a detailed description of System Management Mode (SMM). It includes the following topics:

- What Falls Under the Heading of System Management?
- The Genesis of SMM

- SMM Has Its Own Private Memory Space
- The Basic Elements of SMM
- A Very Simple Example Scenario
- How the Processor Knows the SM Memory Start Address
- Normal Operation, (Including Paging) Is Disabled
- The Organization of SM RAM
- Entering SMM
- Exiting SMM
- Caching from SM Memory
- Setting Up the SMI Handler in SM Memory
- Relocating the SM RAM Base Address
- SMM in an MP System
- SM Mode and Virtualization

The Next Chapter

The next chapter provides a detailed description of the Machine Check Architecture (MCA):

- The MCA Elements.
- The Global Registers.
- The Composition of a Register Bank.
- The Error Code.
- Cache Error Reporting.
- MC Exception Is Generally Not Recoverable.

What Falls Under the Heading of System Management?

The types of operations that typically fall under the heading of System Management are power management and management of the system's thermal environment (e.g., temperature monitoring in the platform's various thermal zones and fan control). It should be stressed, however, that system management is not necessarily limited to these specific areas.

The following are some example situations that would require action by the SM handler program:

- A laptop chipset implements a timer that tracks how long it's been since the hard drive was last accessed. If this timer should elapse, the chipset generates an SMI (System Management Interrupt) to the processor to invoke the SM handler program. In the handler, software checks a chipset-specific status register to determine the cause of the SMI (in this case, a prolonged ces-

sation of accesses to the hard drive). In response, the SM handler issues a command to the hard disk controller to spin down the spindle motor (to save on energy consumption).

- A laptop chipset implements a timer that tracks how long it's been since the keyboard and/or mouse was used. If this timer should elapse, the chipset generates an SMI to the processor to invoke the SM handler program. In the handler, software checks a chipset-specific status register to determine the cause of the SMI (in this case, a prolonged cessation of user interaction). In response, the SM handler issues a command to the display controller to dim or turn off the display's backlighting (to save on energy consumption).
- In a server platform, the chipset or system board logic detects that a thermal sensor in a specific zone of the platform is experiencing a rise in temperature. It generates an SMI to the processor to invoke the SM handler program. In the handler, software checks a chipset-specific status register to determine the cause of the SMI (in this case, a potential overheat condition). In response, the SM handler issues a command to the system board's fan control logic to turn on an exhaust fan in that zone.

The Genesis of SMM

Intel first implemented SMM in the 386SL processor and it has not changed very much since then. While it was not present in the earlier 486 models, it was implemented in all of the later models of the 486 and in all subsequent x86 processors. SMM is entered by generating an SMI (System Management Interrupt) to the processor. Prior to the P54C version of the Pentium, the chipset could only deliver the interrupt to the processor by asserting the processor's SMI# input pin. Starting with the P54C (which was the first IA-32 processor to incorporate the Local APIC) and up to and including the Pentium III, the chipset could also deliver the interrupt to the processor by sending an SMI IPI (Inter-Processor Interrupt) message to the processor over the 3-wire APIC bus. With the advent of the Pentium 4, the 3-wire APIC bus was eliminated and IPIs (including the SMI IPI) are sent to and from a logical processor by performing a special memory write transaction on the processor's external interface.

With the advent of the P54C processor, SMM was enhanced to include the IO Instruction Restart feature (described in this chapter).

The base address of the area of memory assigned to System Management Mode (SMM) has a default value of 30000h. While it could be reprogrammed on the earlier IA-32 processors, the newly-assigned address had to be aligned on an address that was evenly divisible by 32K. Starting with the Pentium Pro, this constraint was eliminated.

SMM Has Its Own Private Memory Space

Prior to the generation of an SMI to the logical processor, the chipset directs all memory accesses generated by the logical processor to system RAM memory:

- When interrupted by an SMI, the logical processor signals to the chipset that all subsequent memory accesses generated by the logical processor are to be directed to a special, separate area of memory referred to as SM RAM.
- Upon concluding the execution of the SM handler program, the logical processor signals to the chipset that all subsequent memory accesses generated by the logical processor are to be directed to system RAM memory rather than SM RAM.

The platform vendor's implementation of SM RAM can be up to 4GB in size.

The Basic Elements of SMM

The following is a list of the basic elements associated with SMM:

- The processor's SMI# input.
- The APIC SMI IPI message.
- The chipset/system board logic responsible for monitoring conditions within the platform that might require an invocation of the SM handler program.
- Chipset's ability to assert SMI# to invoke the SMI handler.
- The chipset's ability to send an SMI IPI message to the logical processor to invoke the SMI handler.
- The Resume (RSM) instruction which must always be the last instruction executed in the SM handler.
- The SM RAM area.
- The logical processor's context state save/restore area (i.e., data structure) in SM memory.
  - 512-bytes for an IA-32 processor.
  - 1024-bytes for an Intel 64 processor.
- The SMI Acknowledge message was added to the message repertoire of the Special transaction.
- On processors that utilize the FSB external interface, the processor's SMMEM# output (also referred to as the EFX4# output).
- The chipset's ability to discern when the processor is addressing regular RAM memory versus SM RAM memory. On processors that utilize the FSB external interface, it does this by monitoring for the processor's issuance of the SMI Acknowledge message and whether or not the processor is asserting the SMMEM# signal during a processor-initiated memory transaction.

33 Machine Check Architecture (MCA)

The Previous Chapter

The previous chapter provided a detailed description of System Management Mode (SMM). It included the following topics:

- What Falls Under the Heading of System Management?
- The Genesis of SMM
- SMM Has Its Own Private Memory Space
- The Basic Elements of SMM
- A Very Simple Example Scenario
- How the Processor Knows the SM Memory Start Address
- Normal Operation, (Including Paging) Is Disabled
- The Organization of SM RAM
- Entering SMM
- Exiting SMM
- Caching from SM Memory
- Setting Up the SMI Handler in SM Memory
- Relocating the SM RAM Base Address
- SMM in an MP System
- SM Mode and Virtualization

This Chapter

This chapter provides a detailed description of the Machine Check Architecture (MCA):

- The MCA Elements.
- The Global Registers.

- The Composition of a Register Bank.
- The Error Code.
- Cache Error Reporting.
- MC Exception Is Generally Not Recoverable.

The Next Chapter

The next chapter provides a complete description of the Local and IO APICs. It includes:

- A Short History of the APIC's Evolution.
- Before the APIC.
- MP Systems Need a Better Interrupt Distribution Mechanism.
- Detecting Presence / Version / Capabilities of Local APIC.
- Local APIC's Initial State.
- Enabling / Disabling the Local APIC.
- Mode Selection.
- The Local APIC Register Set.
- Local APIC ID Assignments and Addressing.
  - ID Assignment in xAPIC Mode.
  - ID Assignment in x2APIC Mode.
  - Local APIC Addressing.
  - Lowest-Priority Delivery Mode.
- Local APIC IDs Are Stored in the MP and ACPI Tables.
- Accessing the Local APIC ID.
- An Introduction to the Interrupt Sources.
- Introduction to Interrupt Priority.
- Task and Processor Priority.
- IO / Local APICs Cooperate on Interrupt Handling.
- Message Signaled Interrupts (MSI).
- Interrupt Delivery from Legacy 8259a Interrupt Controller.
- SW-Initiated Interrupt Message Transmission.
- x2APIC Mode's Self IPI Feature.
- Locally Generated Interrupts.
  - The Local Vector Table.
  - Local Interrupt 0 (LINT0).
  - Local Interrupt 1 (LINT1).
  - The Local APIC Timer.
  - The Performance Counter Overflow Interrupt.
  - The Thermal Sensor Interrupt.
  - Correctable Machine Check (CMC) Interrupt.
  - The Local APIC's Error Interrupt.

- The Spurious Interrupt Vector.
- Boot Strap Processor (BSP) Selection.
- How the APs are Discovered and Configured.

Why This Subject Is Included

When first introduced in x86 processors, the Machine Check Architecture error logging facility was not architecturally defined. Rather, it was a processor-specific addition to the Pentium and Pentium Pro processors with no guarantee that it would be implemented in subsequent processors or, if it was, that it would be implemented in the same manner. It was only with the advent of the Pentium 4 that it was defined as part of the x86 software architecture.

MCA = Hardware Error Logging Capability

The Machine Check Architecture facility consists of a set of error logging registers and the Machine Check exception. During a power up session, a logical processor may experience one or more hardware-related errors internally or on its external interface. Such errors can be divided into two basic categories:

- Soft errors that are automatically corrected by the processor hardware.
- Hard errors that cannot be automatically corrected.

It is expected that the OS will start a daemon that runs in background and periodically examines the MCA registers to determine if any soft errors have been logged since the last time the registers were examined. If so, the application snapshots the errors in a non-volatile storage medium (e.g., on the hard drive or in flash memory) and then clears the errors from the register set (the registers are then available to record any additional errors that may occur in the future).

If a hard error is detected and the Machine Check exception has been enabled, the logical processor records the error in the register set and also generates a Machine Check exception [see “Machine Check Exception (18)” on page 778 for a detailed description] to report it. In the Machine Check exception handler, the error recorded in the register set is read, recorded in the non-volatile storage medium, and possibly displayed on the console. Generally speaking, software cannot recover from most hard errors.

The MCA Elements

The Machine Check Architecture first appeared in the Pentium in rudimentary form (consisting of only two registers), but was greatly expanded with the advent of the Pentium Pro.

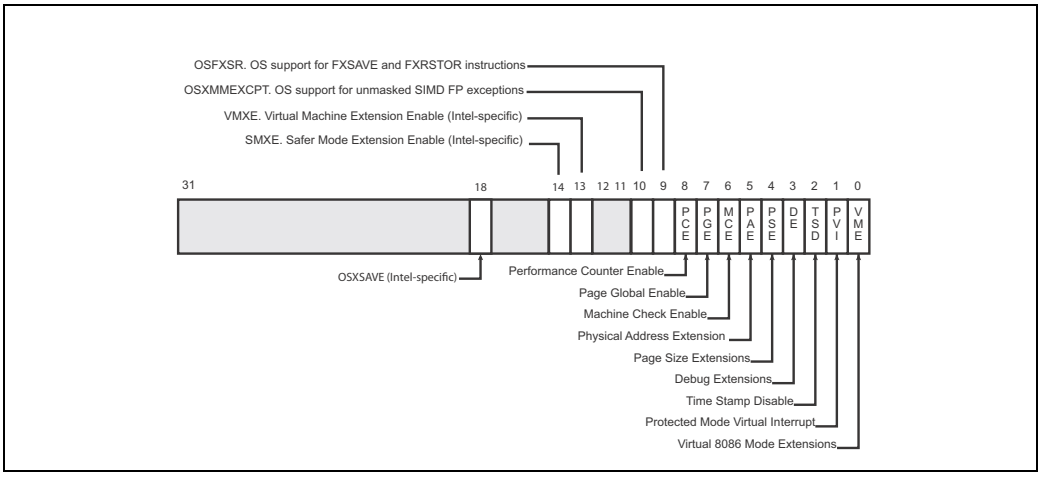
The MCA actually consists of two capabilities (detected by performing a CPUID request type 1 and checking the returned EDX capabilities bit mask): the ability to generate the Machine Check exception and the presence of the MCA register set.

The Machine Check Exception

Although it is optional whether or not a logical processor possesses the ability to generate a Machine Check exception when an uncorrectable hardware error has been detected, all current-day processors support this capability. Whether or not a processor supports this ability is indicated by executing a CPUID request type 1 and checking the EDX[MCE] bit. If it supports the generation of the Machine Check exception, this capability is enabled by setting CR4[MCE] = 1 (see Figure 33-1 on page 1210).

On the P6 processors, the Pentium 4 (and its Xeon and Celeron derivatives), and the Pentium M (and, to the author's knowledge, current-day processors), the Machine Check exception is not recoverable. The interrupted program cannot be safely resumed.

Figure 33-1: Machine Check Exception Enable/Disable Bit (CR4[MCE])



34 The Local and IO APICs

The Previous Chapter

The previous chapter provided a detailed description of the Machine Check Architecture:

- The MCA Elements.
- The Global Registers.
- The Composition of a Register Bank.
- The Error Code.
- Cache Error Reporting.
- MC Exception Is Generally Not Recoverable.

This Chapter

This chapter provides a complete description of the Local and IO APICs. It includes:

- A Short History of the APIC's Evolution.
- Before the APIC.
- MP Systems Need a Better Interrupt Distribution Mechanism.
- Detecting Presence / Version / Capabilities of Local APIC.
- Local APIC's Initial State.
- Enabling / Disabling the Local APIC.
- Mode Selection.
- The Local APIC Register Set.
- Local APIC ID Assignments and Addressing.
  - — ID Assignment in xAPIC Mode.
  - — ID Assignment in x2APIC Mode.
  - — Local APIC Addressing.
  - — Lowest-Priority Delivery Mode.
- Local APIC IDs Are Stored in the MP and ACPI Tables.

- Accessing the Local APIC ID.
- An Introduction to the Interrupt Sources.
- Introduction to Interrupt Priority.
- Task and Processor Priority.
- IO/Local APICs Cooperate on Interrupt Handling.
- Message Signaled Interrupts (MSI).
- Interrupt Delivery from Legacy 8259a Interrupt Controller.
- SW-Initiated Interrupt Message Transmission.
- x2APIC Mode's Self IPI Feature.
- Locally Generated Interrupts.
  - The Local Vector Table.
  - Local Interrupt 0 (LINT0).
  - Local Interrupt 1 (LINT1).
  - The Local APIC Timer.
  - The Performance Counter Overflow Interrupt.
  - The Thermal Sensor Interrupt.
  - Correctable Machine Check (CMC) Interrupt.
  - The Local APIC's Error Interrupt.
- The Spurious Interrupt Vector.
- Boot Strap Processor (BSP) Selection.
- How the APs are Discovered and Configured.

APIC and the IA-32 Architecture

The APIC's (Advanced Programmable Interrupt Controller's) role has been central to the x86 platform's inter-device communication scheme for many years. In addition to providing a communication channel between device adapters and their respective drivers, the APICs allow program threads running on different logical processors to communicate with each other by passing IPI (Inter-Processor Interrupt) messages to each other. Until the advent of the x2APIC architecture, however, the Local APIC's register set and operational characteristics were considered design-specific and outside the scope of the IA-32 architecture. With the introduction of the x2APIC architecture (in the Core i7 processor), the Local APIC's register set is accessible as a set of architecturally-defined MSRs. Even prior to the introduction of the x2APIC feature, however, the Local and IO APICs played a central role in system design. For that reason, this chapter provides a detailed description of the Local and IO APIC operation in both x2APIC Mode as well as the earlier APIC modes of operation.

Definition of IO and Local APICs

Basic definitions of the Local and IO APICs may be found in “APIC” on page 19.

Hardware Context Is Essential

In order to adequately explain the genesis and functionality of the Local and IO APIC modules, it is necessary to have some understanding of their role within the hardware platform. Towards this end, this chapter, where applicable, describes the hardware ecosystem within which the Local and IO APICs fulfill such an important role.

A Short History of the APIC’s Evolution

The following is a *very* short history of the APIC.

APIC Introduction

The APIC was first introduced as a separate, stand-alone chip, the 82489DX. The Local APIC was first introduced in the P54C version of the Pentium.

Pentium Pro APIC Enhancements

The Pentium Pro implemented the following improvements to the Local APIC:

- **Performance Counter Interrupt**. The APIC can be enabled to generate an interrupt if a Performance Counter generates an overflow when incremented. The Performance Counter Overflow entry (see “The Performance Counter Overflow Interrupt” on page 1368) was added to the Local APIC’s Local Vector Table (LVT register set) to support this feature.
- **APIC Base MSR added**. While the memory address range associated with the Local APIC’s register set was hardwired on the Pentium (the base address of the 4KB range was hardwired at FEE00000h), the APIC_BASE MSR added (see Figure 34-11 on page 1258) in the Pentium Pro permits the programmer to specify (in APIC_BASE[35:12]) the register set’s base address starting on any 4KB-aligned address range in the logical processor’s physical memory address space.

- **Software Enable/Disable added**. The Pentium processor's Local APIC could not be enabled or disabled under software control. This could only be accomplished via hardware when the processor sampled an input on the trailing-edge of reset. Starting with the Pentium Pro, the APIC_BASE[EN] bit can be used by software for this purpose.
- **BSP bit added**. The APIC_BASE[BSP] bit is a read-only bit that *remembers* whether the logical processor was selected as the Boot Strap Processor or as an Application Processor (AP) at startup time. See "Boot Strap Processor (BSP) Selection" on page 1378 for more information on the BSP selection process.
- **APIC access propagation deleted**. When software executing on a Pentium performed a load or a store targeting the processor's Local APIC register set, the memory access was also propagated out onto the processor's external interface. This was eliminated with the advent of the P6 processor family.
- **Illegal Register Address error bit added** to the Local APIC's Error Status register (see Figure 34-26 on page 1302).
- **Remote Register Read capability was eliminated**.
- **SMI delivery added**. The ability to deliver an SMI to a logical processor via an Inter-Processor Interrupt (IPI) message was added.

The Pentium II and Pentium III

No enhancements were made to the APIC architecture in the Pentium II and Pentium III processors.

Pentium 4 APIC Enhancements: xAPIC

To differentiate the revised APIC architecture introduced with the advent of the Pentium 4 from the old APIC architecture, it is referred to as the xAPIC architecture. For the remainder of this chapter, the earlier APIC architecture is referred to simply as the *APIC architecture*, or the *legacy APIC architecture*. The xAPIC architecture introduced the following improvements:

- **Thermal Sensor interrupt added** (see "The Thermal Sensor Interrupt" on page 1370).
- **APIC ID Register enhanced**. In the P6 and Pentium, the APIC ID field was 4-bits, and encodings 0h - Eh could be used to uniquely identify 15 different processors connected to the APIC bus. In the Pentium 4, the xAPIC spec extended the local APIC ID field to 8 bits which can be used to identify up to 255 logical processors in the system.

Live Training Courses on the x86 Architecture

Comprehensive x86 Architecture (32/64-bit)

This lecture-based course describes the entire x86 architecture; everything from Real Mode addressing to 64-bit Mode interrupt handling. This course focuses on the architectural elements of x86, like segmentation, interrupt/exception handling, paging, etc. After completing this course, you will have a much deeper understanding of the x86 architecture.

Topics covered in this course:

- Instruction Set
- Register Sets
- Operating Modes
- Segmentation
- Task Management
- Interrupts and Exceptions
- Paging
- Memory Types
- Virtualization

System Programming for the x86 Architecture

This course teaches the x86 architecture (both 32-bit and 64-bit) through a mix of lectures and hands-on programming labs. All topics are explained in lecture format first and then the students are given programming labs in assembly to reinforce the concepts and to get hands-on experience working with x86 processors at a very low level. This course focuses mainly on the behavior of legacy Protected Mode, Compatibility Mode and 64-bit Mode as these are the modes most commonly used in modern operating systems.

The lab exercises range from printing to the screen using the flat memory model in legacy Protected Mode to setting up an interrupt driven, multitasking 64-bit Mode environment, with paging turned on.

You Will Learn:

- x86 programming basics like an overview of the instruction set, register set and operating modes
- The behavior of segmentation, how it was originally intended to be used and how it is actually used by operating systems today
- How to setup system calls using multiple methods (and benefits / side-effects of each)
- How to setup interrupt service routines for both software and hardware interrupts and implement a rudimentary scheduler
- How to implement paging in both the 32-bit environments as well as the 64-bit environments including using various page sizes

eLearning Course on the x86 Architecture

Intro to 32/64-bit x86 Architecture

MindShare's Intro to x86 eLearning course provides a great overview of the x86 instruction set architecture. It describes the concepts and implementation of the major architectural elements of x86 (paging, interrupt handling, protection, register set, address spaces, operating modes, virtualization, etc.).

You Will Learn:

- The different groupings of x86 instructions and basic instruction formats
- The different address spaces available in x86 and how each can be accessed
- The registers defined in x86 and how they have grown over time
- All the operating modes that x86-based processors can run in and characteristics of each
- The concepts of paging and its base implementation in x86
- How interrupts and exceptions are handled inside the core
- What virtualization is and about the hardware extensions that companies like Intel are adding their processors

What's Included:

- Unlimited access to the x86 eLearning modules for 90 days
- PDF of the course slides (yours to keep, does not expire)
- Q&A capability with the instructor

Benefits of MindShare eLearning:

- ***Cost Effective*** - Get the same information delivered in a live MindShare class at a fraction of the cost
- ***Available 24/7*** - MindShare eLearning courses are available when and where you need them
- ***Learn at Your Pace*** - MindShare eLearning courses are self-paced, so you can proceed when you're ready
- ***Access to the Instructor*** - Ask questions to the MindShare instructor that taught the course