CHIPS

Super386™ DX *Programmer's Reference Manual*

Super386™ DX

Programmer's Reference Manual

CHIPS®

CHIPS

Super386™ DX

Programmer's Reference Manual

Chips and Technologies, Inc.
3050 Zanker Road
San Jose, California 95134
Phone: 408-434-0600

Preface

This manual describes the software architecture of the Chips and Technologies Super386™ DX/DXE processors—38600DX, 38605DX, 38600DXE, and 38605DXE. These processors are software-compatible with the industry-standard 80386 processor. The manual is addressed to experienced assembly-level programmers writing either application or system software. No previous knowledge of the 80386 processor or any similar processor architecture is assumed.

Unless otherwise stated, the term "processor" refers to both the 38600DX/DXE and 38605DX/DXE processors. The descriptions throughout most of the manual assume that the processor is running in its fully featured, 80386-compatible protected mode, which is explained in Chapter 2. The processors support two other modes for 8086 programs: real mode and virtual-8086 mode. The functioning of these modes is explained in the section entitled "Other Processor Modes" in Chapter 4.

Organization

The manual contains four chapters and three appendices:

- Chapter 1, Introduction—Overview of the Super386 processors and a list of their features, including the SuperState V feature of the DXE processors.
- Chapter 2, Programmer's Model—Description of the Super386 processor as a collection of resources available to software. The chapter discusses the three execution modes, the data types directly supported by the instruction set, the organization of external memory and I/O spaces, the registers visible to software, and interrupts and exceptions. The chapter also describes the on-chip instruction cache of the 38605DX and DXE processors.
- Chapter 3, Instruction Set—Overview of the instruction set, organized by function. The chapter discusses operand types, addressing modes, flags, condition codes, and instruction encoding.
- Chapter 4, System Programming—Discussion of such operating-system issues as memory management, protection, I/O access, multitasking, interrupt and exception handling, and processor initialization. The chapter describes the use of execution modes other than the processor's native protected mode. It also describes the SuperState™ V features of the DXE processors.

Application programmers need most of the material in the first three chapters. In these chapters, items of interest only to system programmers are identified as such. System programmers need all the material in the book.

Reference material is provided in three appendices:

- Appendix A, Instruction Set Reference—List of the instructions arranged alphabetically by assembler mnemonic. Gives detailed information on each instruction.
- Appendix B, Quick Reference Tables—Summary lists of opcodes, flag cross references, status flags, condition codes, instruction formats, and timing.
- Appendix C, Special Programming Considerations—Discussion of the effective use of advanced features.

A glossary of acronyms is provided along with an index.

Notations and Conventions

The following notations and conventions are used:

- Processor Names—In general, the terms *processor* and *Super386 processor* apply to all the Super386 DX and DXE processors. When only one of these processors is referred to, or when the 80386 processor is referred to, the processor is named explicitly.
- Byte Quantities—Kilobytes is kB, megabytes is MB, gigabytes is GB.
- Binary and Hexadecimal Numbers—Binary numbers are followed by a b and hexadecimal numbers by an h. Numbers without a suffix are decimal. Thus, 00010001b = 17 = 11h.
- LSB—The least significant bit in the binary representation of a number is bit 0. In diagrams, bit 0 is at the right and the most-significant bit is at the left.
- Little-Endian Format—The Super386 processor is a *little-endian* machine. That is, a multiple-byte quantity is always stored with its least-significant byte at the lowest byte address. In illustrations, words and doublewords are shown with the least significant byte at the right. Byte addresses increase from right to left. As a consequence, strings are shown in reverse order.
- Memory Addresses—In illustrations of data structures in memory, the lowest memory address is at the bottom.
- Addressable Quantities—An 8-bit quantity is referred to as a byte; a 16-bit quantity is a word; and a 32-bit quantity is a doubleword or dword. The precessors described herein use byte addressing, in which memory is accessed as a sequence of addressable bytes.

- Segmented Addressing—When a segmented address contained in a register is mentioned, the acronym for both the segment and the register are shown, separated by a colon. For example, the address of a memory location contained in the data segment (DS) with an offset contained in the EBX register would be written as DS:EBX.
- Bit Ranges—When a range of bits is referred to, the highest and lowest bit numbers are shown, separated by a colon. For example, when the range is bit 15 to bit 9, it is referred to as 15:9.
- Bit Values—Bits can either be *set* or *cleared.* The term *set* means the bit has a binary value of 1. The term *cleared* means the bit has a binary value of 0.
- Reserved Bits—Some bits and bytes in register illustrations are marked *not available.* Do not store or use data at these locations. These bits should be masked out before testing, and the bit states should not be changed when the rest of the register is accessed.

Related Documents

The following related documents are available from Chips and Technologies:

- *Super386™DX/DXE High Performance CMOS Microprocessor Data Book*
- *Super386™DX Performance Test Report*
- *Super386™/SuperMath Compatibility Brief*
- *SuperState V™Architecture Manual.*

In addition to these publications from Chips and Technologies, several commercial books provide special insights and different perspectives on programming with the Super386 processor. Rakesh Agarwal's two-volume book, *80x86 Architecture & Programming*, is an excellent guide for system programmers, with many examples of system routines written in pseudo-code. John Crawford and Patrick Gelsinger's book, *Programming the 80386*, provides another valuable viewpoint. This book is well illustrated and provides pseudo-code examples of common system software routines. Stephen Morse, Eric Isaacson, and Douglas Albert's book, *The 80386/387 Architecture*, is a clearly written text that relates the basic concepts of 80386 architecture to the earlier versions of that architecture.

Contents

CHAPTER 3

CHAPTER 4

APPENDIX A

APPENDIX B

APPENDIX C

List of Figures

List of Tables

CHAPTER 1

Introduction

The Super386 DX/DXE processors provide higher performance than the comparable standard 80386 processors, with which they are code-compatible. Like the 80386 processors, the Super386 processors support multitasking operating systems and are designed for use in computation-intensive applications. They operate faster than standard 80386 processors due to their entirely redesigned internal architecture and unique microcode.

There are currently four Super386 DX/DXE processors:

- 38600DX
- 38600DXE
- 38605DX
- 38605DXE.

These processors are discussed below.

The 38600DX processor is a high-performance, static CMOS implementation of the 80386 DX processor's 32-bit architecture, with hardware support for jump instructions. It is pin-compatible with the 80386 DX processor and is a superset of its functionality.

Processor 38600DXE is identical to the 38600DX but it incorporates the SuperState V feature, a system for power management. This feature works in all modes and makes the 38600DXE processor suitable for low-power applications.

The 38605DX processor has all the features of the 38600DX processor but adds a 512-byte instruction cache. The 144-pin package is a superset of the 38600DX pinout. Systems designed for the 38605DX footprint can also use the 38600DX processor in the same socket.

The 38605DXE features both the 512-byte instruction cache and SuperState V mode for special applications. Two special pins are added to facilitate operation in SuperState V mode: ANMI*, an alternate non-maskable interrupt input, and AADS*, an alternate address space output. See the section entitled "SuperState V Mode" in Chapter 4 for a description of these signals.

In general, the terms *Super386 processor* and *processor* apply to both the 38600DX/DXE and the 38605DX/DXE processors. When only one of these processors is referred to, or when the 80386 processor is referred to, the processor is named explicitly.

Features of All Super386 Processors

Features common to all processors in the Super386 family are:

- 80386 compatibility
- Memory management
- High-performance pipeline
- Advanced CPU clock design
- Static design
- Coprocessor support.

These features are discussed in the following paragraphs.

80386 Compatibility—The Super386 processors are object-code compatible with the standard 80386 processor and support all operating modes supported by the 80386 processor.

Memory Management—The memory management features include segmentation and paging. Segmentation allows programmers to create independent, protected address spaces. Paging makes it possible to use virtual data structures that are larger than the available memory space, by keeping the data partly in memory and partly in a mass-storage device.

High-performance Pipeline—The new pipeline design permits overlapping of instruction execution at CPU clock rates up to 40MHz.

Advanced CPU Clock Design—Systems designers can use a 1x or 2x CPU clock running from 0 to 25, 33, or 40MHz.

Static Design—All on-chip registers, buffers, and instruction cache (38605 only) are fully static, allowing the CPU clock to be stopped without losing data.

Coprocessor Support—For floating-point operations, the Super386 processors support the SuperMath™ coprocessor and standard 80387 coprocessors.

Special Features

Certain new features distinguish the 38605 processor from its predecessors: The 38605 processor prefetches instructions and stores them in a 512-byte instruction cache located on the chip. The processor goes to the cache for the next instruction and only fetches instructions from memory when the next instruction is not in the cache.

Near jump instructions are handled by dedicated hardware, as in the 38600 processor. But in combination with the instruction cache, this jump hardware improves near jump execution speed dramatically: two cycles with the jump hardware and cache versus six cycles without.

The 38600DXE and 38605DXE processors both feature SuperState V Mode. This special mode of operation is designed for power management and device emulation. It is transparent to the normal operating environment, permitting a control program, running at a more priviledged level, that allows the operating system to access the processor for special power management and feature control purposes.

CHAPTER 2

Programmer's Model

The Super386 architecture offers software developers a variety of registers, data structures, and other resources. This chapter describes the organization of memory, mechanisms by which system-level resources are protected from use by application software, and the different modes of instruction execution. It also defines the data types supported by the instruction set, describes the processor registers available to application programs, and introduces the basic types of interrupts and exceptions. The concepts covered in this chapter are referred to throughout this manual. For system programmers, the discussion continues in Chapter 4, "System Programming.

Memory Organization

The processor can directly access up to 4GB of *physical address space*, each byte of which is separately addressable using a 32-bit *physical address*. During each memory access (or for the 38605 processor, each non-cached memory access), a physical address appears on the processor bus. External logic decodes the physical address into control signals for external memory or peripheral devices.

Software does not supply physical addresses directly to the processor. Every instruction that accesses memory supplies instead a *logical* (or *virtual*) *address,* which is translated into a physical address by the processor's memory-management unit.

In the processor's native, fully featured 32-bit mode—called *protected mode*—the address-translation mechanism makes use of translation tables created and maintained by the operating system. Thus, while the physical address space is a simple one-dimensional sequence of bytes, the *logical* organization of the external memory space—the way memory appears to software—can take on more complex forms determined by the operating system.

In particular, the processor supports *segmented memory*, in which a linear one-dimensional memory space is broken up by the operating system into independent linear, unbroken regions called *segments*. In protected mode, each program can have up to 16,384 segments, possibly overlapping, with sizes up to 4GB. Segments can be explicitly assigned to hold code, program stacks, or data. Segmentation can preserve the integrity of program code and data during unanticipated software accesses, such as erroneous or unauthorized access to one program's data or stack by another program. Figure 2-1 illustrates one way in which memory could be organized into segments.

Figure 2-1. *Example of Segmented Memory*

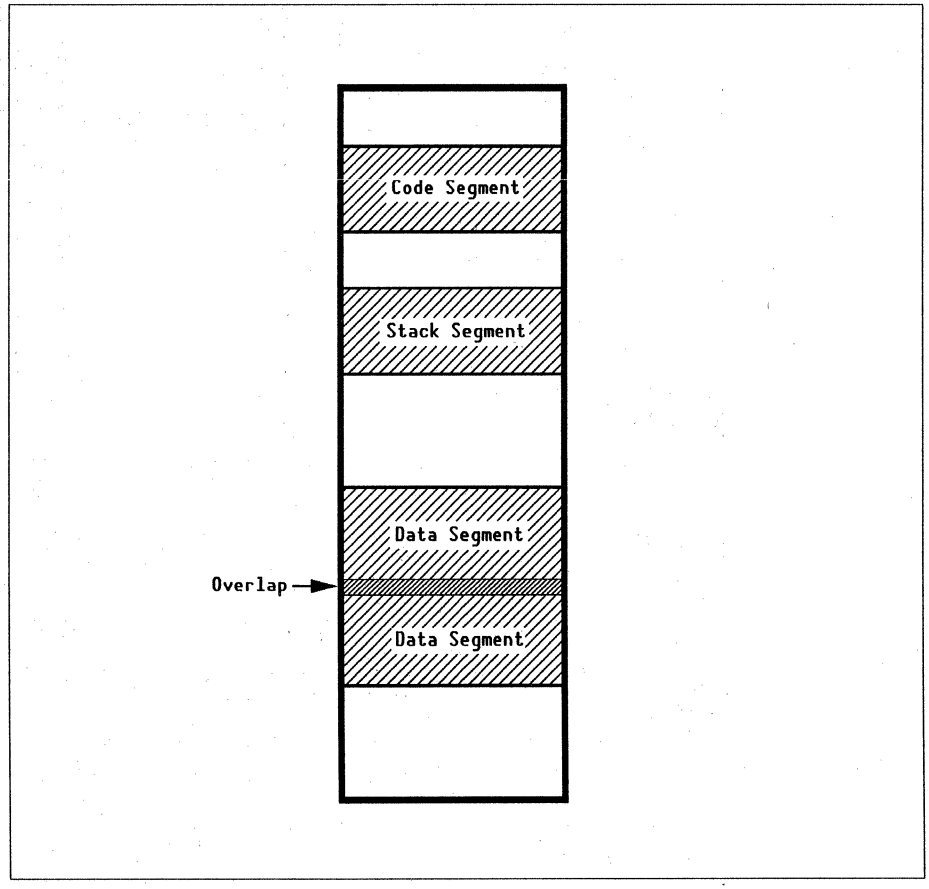
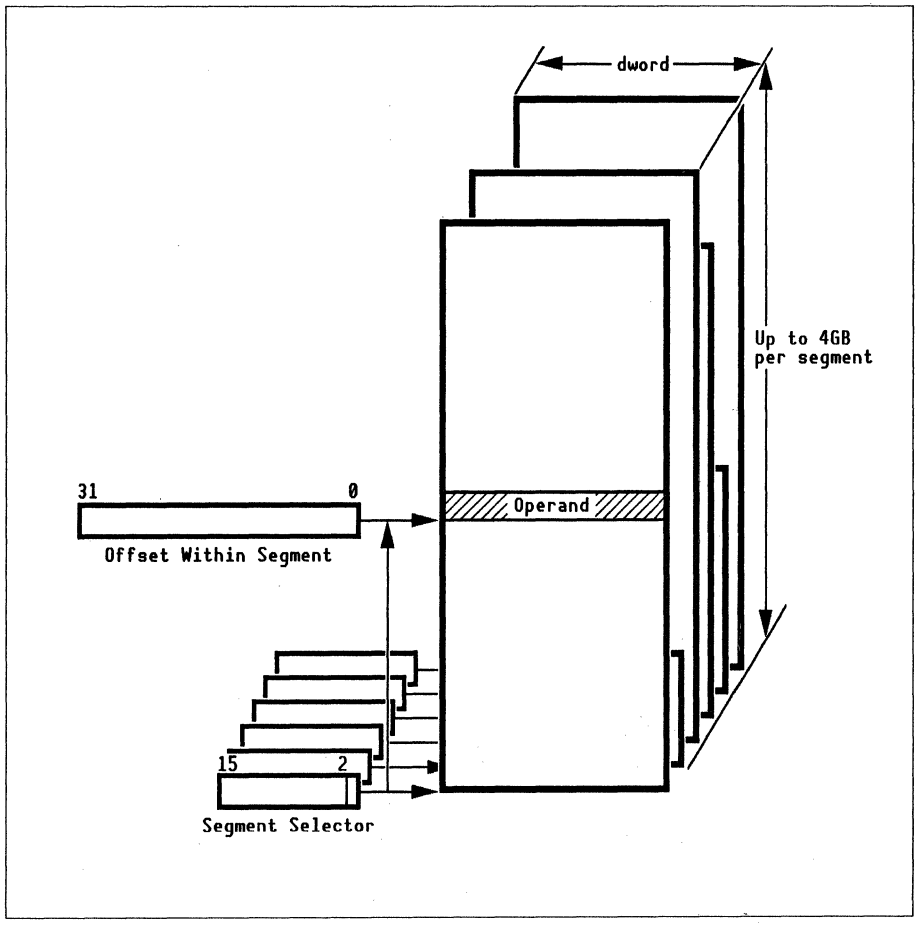


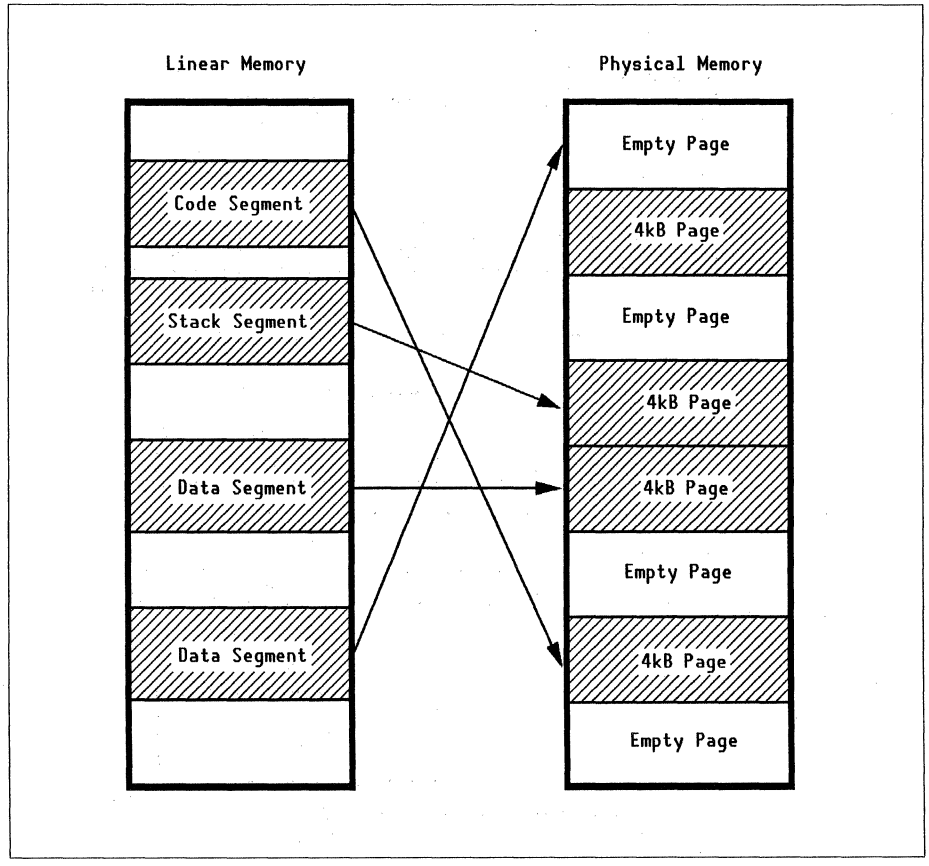
Figure 2-2 shows how a logical address is used to locate an operand in a segmented memory space. One part of the logical address identifies a segment; the other part specifies an offset into that segment. In protected mode, the *segment selector* provides an index into a descriptor table. Data in the descriptor table locates the *base address* of the segment. The *offset* then locates the addressed byte within the segment.

Figure 2-2. *Logical Addressing of Segments*



Paging, which is illustrated in Figure 2-3, is another aspect of memory management. Paging maps linear addresses generated by segmentation into physical addresses in memory. It is a technique for simulating a large external memory by swapping data between RAM and a mass-storage device such as a disk. Data is swapped in units of 4kB called *pages*. The operating system keeps track of which pages are in RAM at any given time and which are on a disk. A request for data currently held on disk causes an exception. The service routine for the exception loads the page with the requested data into RAM, swapping some other page out to disk if necessary.

Figure 2-3. *Linear Addressing of Pages*



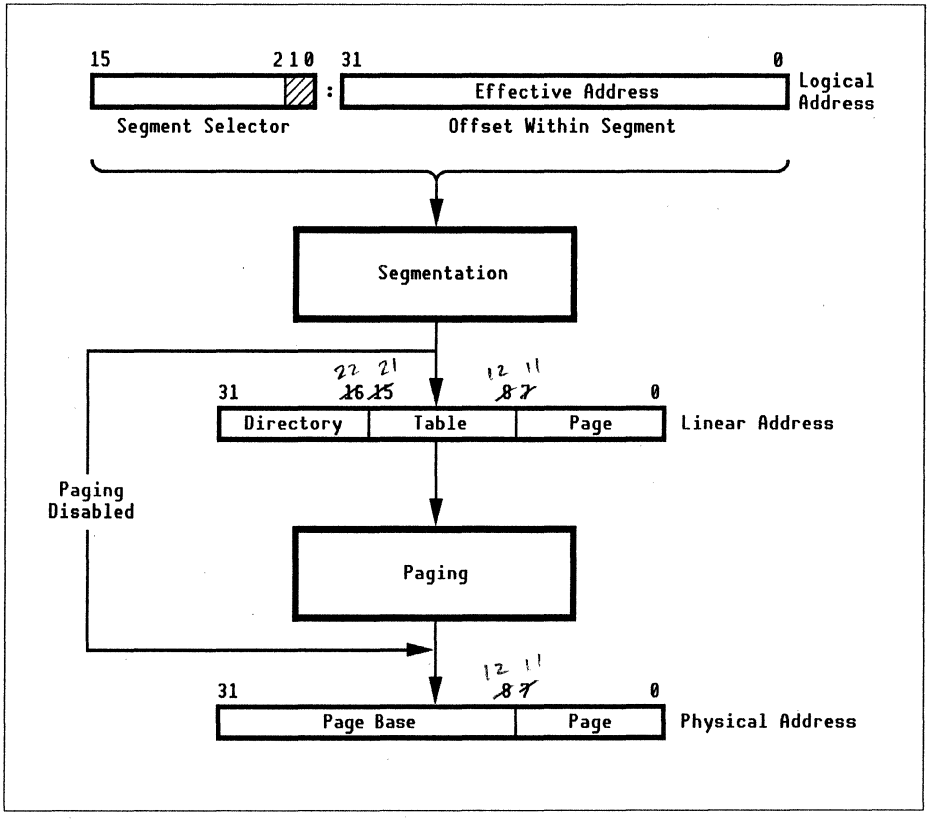
Address Translation

Address translation is part of the processor's segmentation and paging mechanisms. It has two stages, as illustrated in Figure 2-4.

Segmentation—In the segmentation stage, the processor's segmentation unit translates the logical address supplied by software into a linear address, which specifies the location of a byte in a one-dimensional linear address space.

Paging—If paging is enabled, the linear address undergoes further translation. This stage of address translation is carried out with information contained in page directories and page tables. These data structures reside in memory and are maintained by the operating system. If paging is disabled, or is unavailable in the processor's current execution mode, the linear address is used as the physical address.

Figure 2-4. *Overview of Address Translation*



Addressing Segmented Memory

While paging is transparent to application programs (except for occasional delays when data needs to be swapped), segmentation is an everyday fact of life for even the most casual assembly-language programmer. Every instruction that accesses memory must indicate a segment for the intended access as well as an offset into that segment.

At any given time, up to six segment selectors reside on-chip in the processor's six *segment registers*. A memory reference in an assembly language instruction must specify—either explicitly or by default—one of these registers. In protected mode, the high-order 13 bits in a segment register specify an offset into a *segment descriptor table* (GDT or LDT), which in turn locates the segment. Segment descriptors are described further in the next section.

Memory references also have an offset into the selected segment. This offset, or *effective address*, can be specified in various ways known as the *addressing modes* of the processor. Basically, up to three components can be added together to form the offset: the contents of a specified base register, the scaled contents (multiplied by 1, 2, 4, or 8) of a specified *index register*, and a constant value called a *displacement*. The various addressing modes support complex data structures typically used in high-level languages.

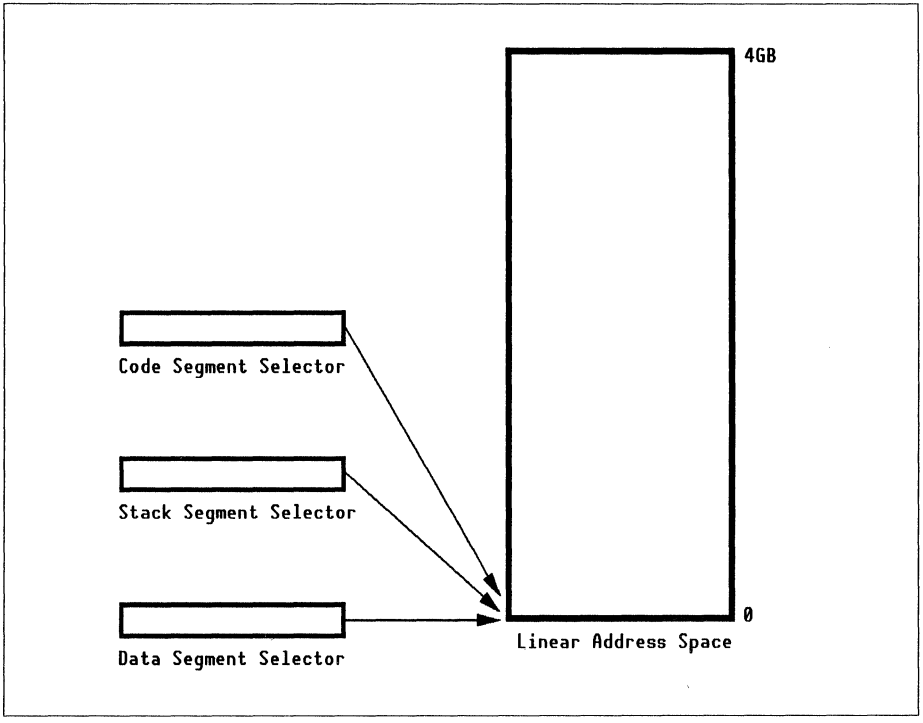
The ways in which instructions address memory operands are discussed in detail in Chapter 3, "Instruction Set Overview."

Descriptor Tables and Memory Models

In protected mode, the offset into a segment descriptor table that the segment selector provides locates an 8-byte *segment descriptor* associated with the selector. The segment descriptor contains information about the corresponding segment, including its base address (in the linear address space) and its size. This information is used in logical-to-linear address translations. Segment descriptors, along with descriptors of other kinds, are maintained by the operating system in data structures called *descriptor tables*. By controlling the contents of the descriptor tables, therefore, the operating system controls the logical organization of memory.

The simplest memory organization is the *flat model*, in which all segment descriptors point to the same base address and specify the same segment size. Figure 2-5 illustrates a flat memory organization.

Figure 2-5. *Flat Memory Model*



While there is no way to disable the processor's segmentation mechanism, using the flat model achieves the same result: memory is accessed as a single range of linear addresses. The size of this range can be up to the 4GB maximum or restricted to the actual size of the external memory. The latter approach has the advantage that out-of-range addresses will be trapped by the segmentation unit. See the section entitled "Protection."

Segmented memory models, on the other hand, can be quite complex. Each application can be given its own descriptor table, defining up to 16,384 distinct segments. Each of these segments can be of any size up to the 4GB maximum. Some segments can be reserved for a given application, while others are shared. The operating system can map segments to overlapping ranges in the linear address space.

Figure 2-6 illustrates a moderately complex segmentation strategy in which each of two applications has multiple data segments. The two applications also share a data segment.

Figure 2-6. *Segmentation Strategy: Example 1*

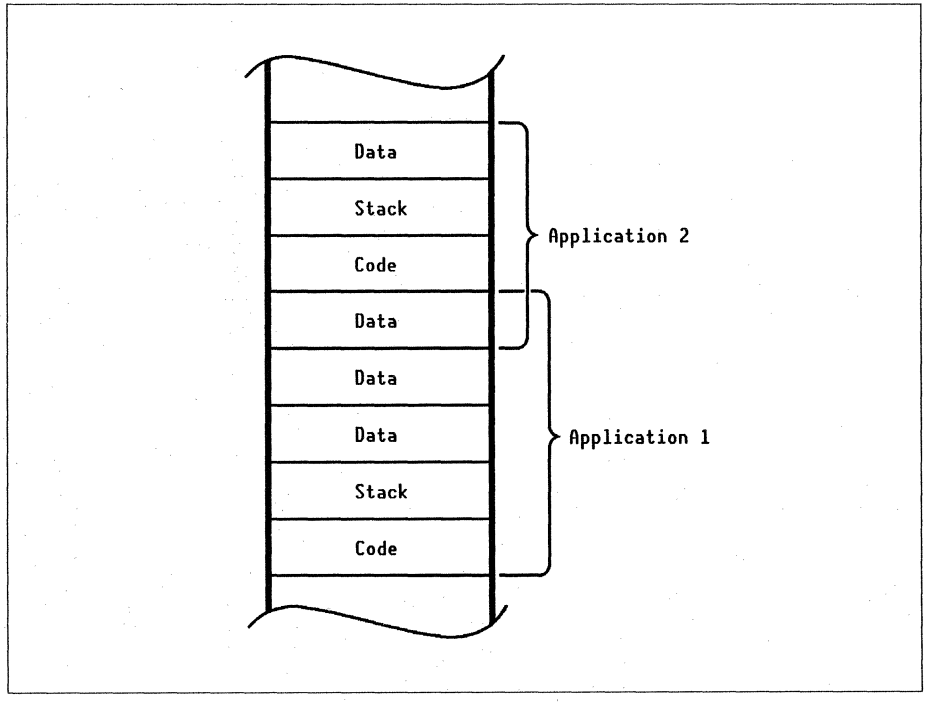
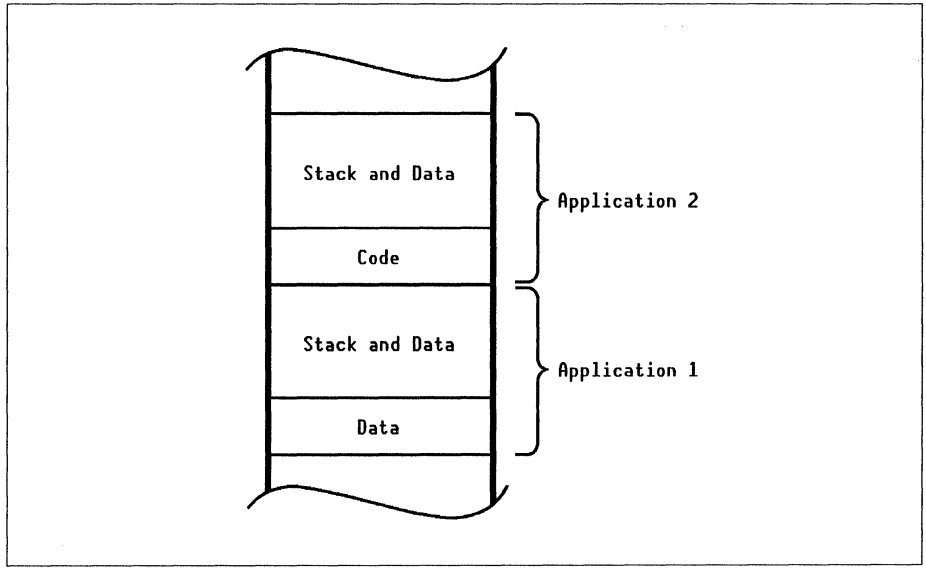


Figure 2-7 illustrates a simpler segmentation strategy, in which each application has a single segment to hold its stack and data. This arrangement has the advantage that 32-bit pointers can be used to access data (instead of 48-bit pointers). On the other hand, the stack is not prevented from growing down into the region where the program stores data. See the sections entitled "Resource Protection" and "Stack Operations" for futher information.

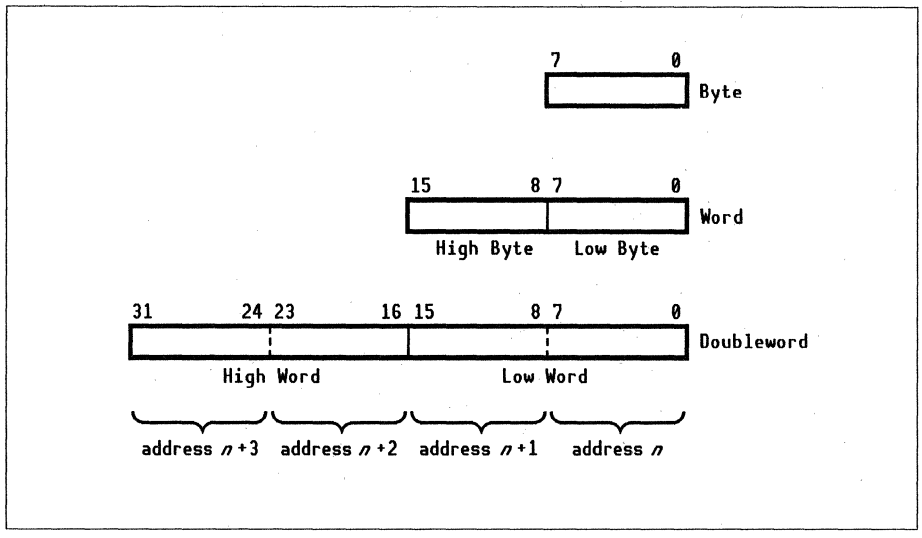
Figure 2-7. *Segmentation Strategy: Example 2*



Storing Data in Memory

Data items held in memory or in processor registers can be of several different lengths. A *byte* is an ordered sequence of 8 bits; it is the smallest addressable quantity. A *word* is a sequence of 16 bits. A *doubleword* (or *dword*) is a sequence of 32 bits. See Figure 2-8.

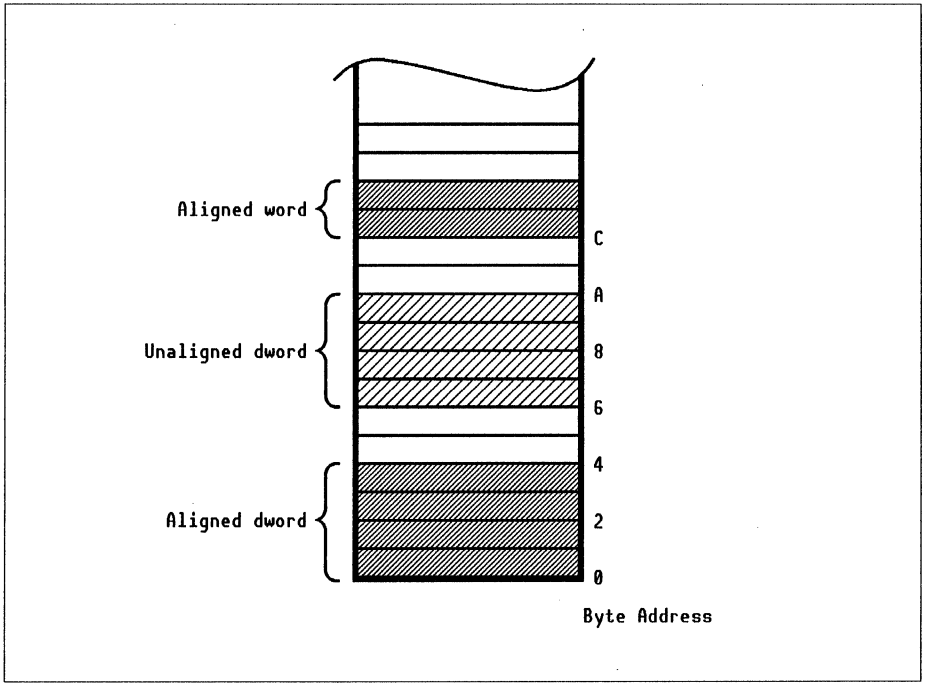
Figure 2-8. *Representation of Data in Memory*



The processor uses *little-endian* encoding, in which a multi-byte quantity is stored with its least-significant byte at the lowest byte address. In the illustration, words and doublewords are shown with the least significant byte at the right. Byte addresses increase from right to left. As a consequence, numerical data reads normally, with the most significant hexadecimal digits appearing at the left. Strings, however, read in reverse order.

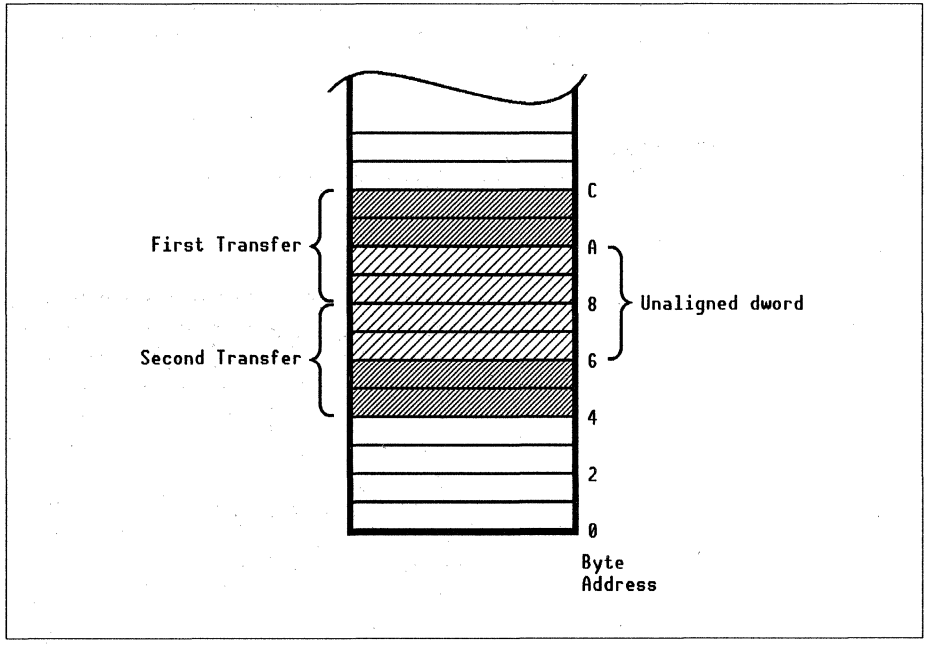
Multi-byte quantities in memory are always addressed using the byte address of the least-significant byte. A memory word is said to be aligned when this address is an even number. A dword is aligned when the address of its least-significant byte is divisible by 4. In general, any 2^n-byte quantity is aligned if its address is a multiple of its size in bytes. See Figure 2-9.

Figure 2-9. *Data Alignment*



Physically, each external memory access transfers between one and four bytes of an aligned dword between the processor and memory. To access a multi-byte quantity that crosses a dword boundary, the processor performs multiple transfers. For example, to transfer an unaligned dword requires two transfers, as illustrated in Figure 2-10. While such accesses are handled automatically by hardware, they do require extra bus cycles with a consequent penalty in performance.

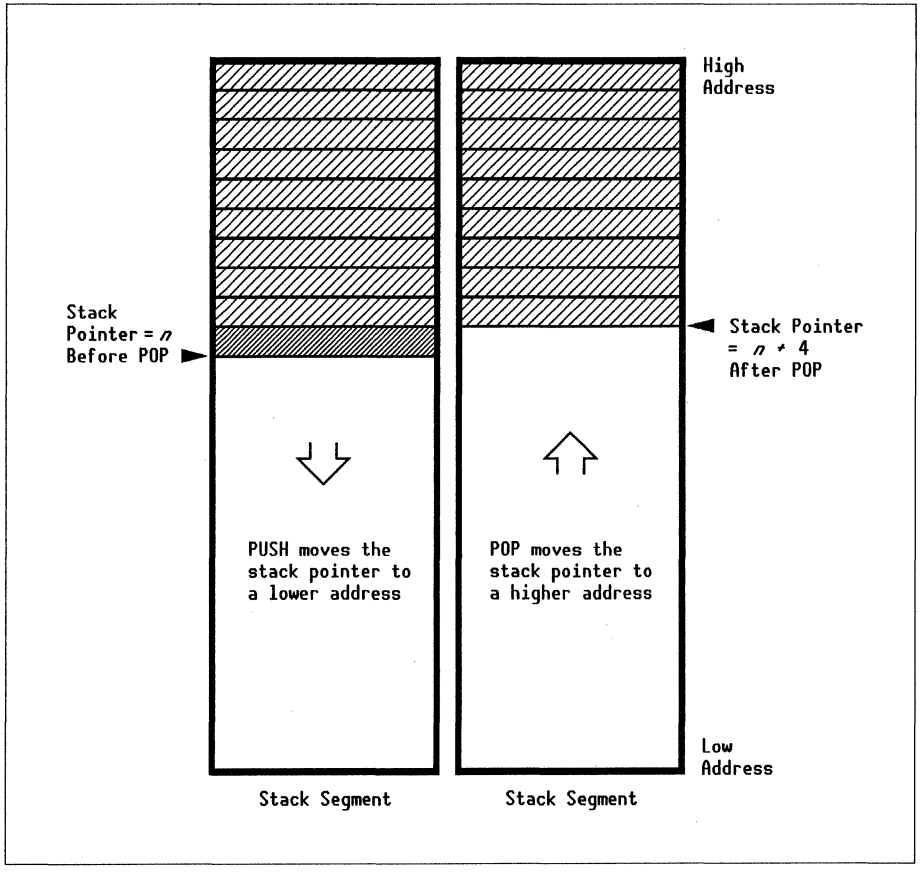
Figure 2-10. *Unaligned Accesses*



Stack Operations

Several instructions directly manipulate the *program stack* (or simply *stack*). Stacks implement a last-in first-out (LIFO) data structure. They are typically used in situations that require nested storage such as subroutine calls and the evaluation of complex expressions. Each stack can be contained in a separate memory segment. One stack—the *current stack*—is directly addressable at any given time. Its segment selector is the value in the Stack Segment (SS) register. The location of the current *top of stack* (the last operand written to the stack) is the value in the Stack Pointer (ESP) register. The ESP register specifies an offset into the current stack segment. Data can be appended to the current stack using a PUSH instruction, or removed from the stack using a POP instruction. When data is appended, the stack grows toward lower memory addresses in the linear address space, as shown in Figure 2-11 for 32-bit addresses.

Figure 2-11. *Stack Organization, 32-Bit Operation*



A procedure call automatically pushes its return address onto the stack. Upon return from the procedure, the address is popped. The last-in-first-out allocation rule makes it easy and efficient to handle nested subroutines, even when these are recursive or re-entrant. A series of CALL instructions will leave a sequence of addresses on the stack. The first RET instruction thus finds the return address of the most recent CALL at the top of the stack. The stack can also be used to pass parameters to a subroutine, or to store a subroutine's local variables.

The registers used to implement stack operations are discussed in more detail in the section entitled "Registers." Details of the PUSH and POP instructions are discussed in Appendix A, "Instruction Set Reference."

Input/Output

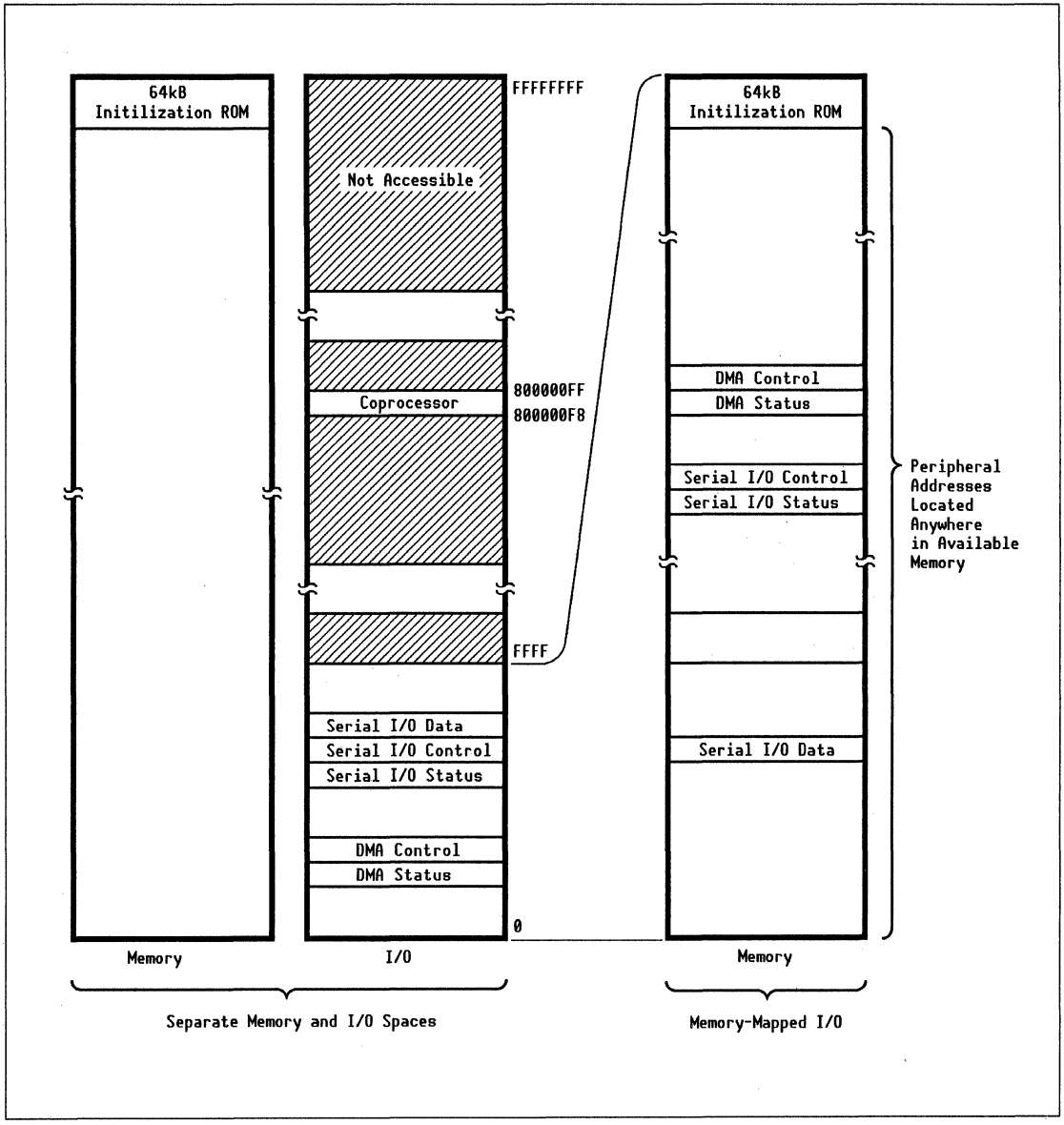
Depending on system implementation, I/O peripheral devices can be accessed in one of two address spaces: I/O space and memory-mapped I/O.

I/O Space—In this arrangement, the control, status, and data ports for peripheral devices are located in an addressable space that is separate from the memory space. Special I/O instructions are used to transfer data between these ports and the processor registers or memory.

Memory-Mapped I/O—In memory-mapped I/O, the control, status, and data ports for peripheral devices share the normal memory space with all other memory segments. Accesses to these I/O addresses work in the same way as normal memory accesses.

Figure 2-12 shows these two alternative arrangements.

Figure 2-12. *I/O Space and Memory-Mapped I/O*



I/O Space

The I/O space is a 64kB linear address space beginning at I/O address 0. Ports can be 1, 2, or 4 bytes wide. Architecturally and physically, I/O space is separate from memory space. Separation of memory and I/O space offers the most reliable system protection: the I/O space has its own protection mechanisms, separate from those applied to the memory space. For example, the system design can prevent reads and writes to I/O space from being captured by a cache. When a separate I/O space is used, however, it can only be accessed by the I/O instructions IN, INS, OUT, and OUTS.

Memory-Mapped I/O

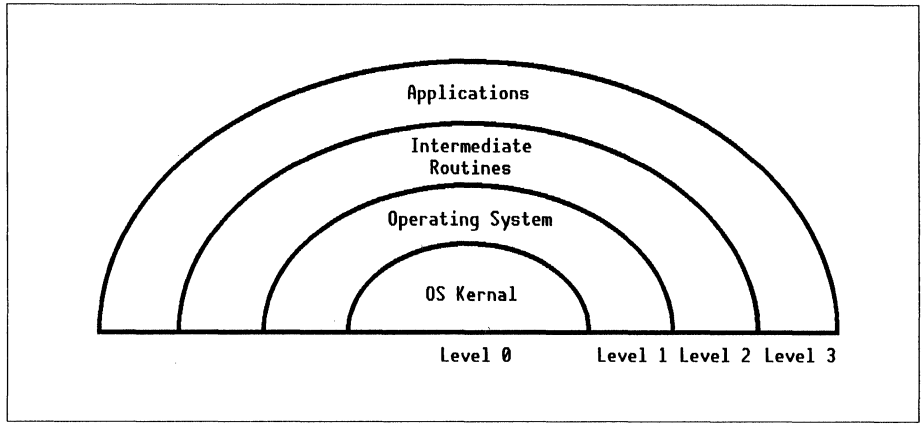
The chief advantage of memory-mapped I/O is that the general-purpose arithmetic and logical instructions, which operate on memory-space operands, can also be used for I/O. For example, memory-mapped I/O allows application software to set bits in a peripheral register without passing the contents of the peripheral register through a processor register.

Resource Protection

Every memory segment has an associated privilege level represented as a number between 0 (most privileged) and 3 (least privileged). The privilege level of the code segment from which instructions are currently being fetched is called the *current privilege level* (CPL). In general, protected resources can be used only by sufficiently privileged code.

In a typical arrangement, the operating system kernel runs at level 0 and the rest of the operating system runs at level 1. Applications run at level 3, and level 2 is left free for special-purpose code requiring an intermediate degree of privilege. Figure 2-13 illustrates this arrangement.

Figure 2-13. *Privilege Levels*



The operating system can implement protections for three types of resources:

- Privileged instructions
- Memory
- I/O.

These resources are discussed in the following paragraphs.

Privileged Instructions—Privileged instructions are machine instructions that can be used only by code at privilege level 0. Examples are instructions that explicitly modify system control registers.

Memory—The tables used in address translation (descriptor tables, page directories, and page tables) contain bits that restrict access to individual segments and pages. Attempted memory accesses by insufficiently privileged code are trapped.

I/O—The use of I/O instructions and ports can be restricted by the operating system to code of a given privilege level (or better). Global protection is applied through the two-bit I/O privilege level (IOPL) flag in the EFLAGS register. The IOPL specifies the minimum privilege level required to execute I/O instructions. Port-level protection is provided by an operating system data structure called the I/O permission bitmap (IOPB), which controls access to individual I/O ports based on privilege level.

Access rules for the various protected resources are discussed in detail in Chapter 4, "System Programming."

Execution Modes

The processor has three mutually exclusive modes of instruction execution that are selectable by system software:

- Protected mode
- Real mode
- Virtual-8086 mode.

These modes provide full 32-bit processing, protection, and virtual-memory features for newly written code while ensuring compatibility with code written for 16-bit processors. The modes also provide support for mixing 16-bit and 32-bit code.

Protected Mode—In protected mode, all of the processor's segmentation, paging, protection, and multitasking capabilities are available. Programs written for protected mode on the 80386 and 80286 processors can be run in protected mode on a Super386 processor. Maximum linear memory size is 4GB, and default operand size can be 16 or 32 bits.

Real Mode—Real mode is the 8086 real-address emulation mode. Maximum memory size (1MB), default operand size (16-bit), address generation, and interrupt handling are nearly identical to the 80286 real mode. Instruction prefixes allow use of 32-bit operands, giving full use of the 32-bit registers. All code runs at privilege level 0. Protected segmentation and paging are not available.

Virtual-8086 Mode—In virtual 8086 mode the processor generates 8086 real-mode addresses, but with the virtual-memory paging capabilities of protected mode. Like real mode, virtual-8086 mode has a maximum memory size of 1MB. Programs run as tasks. The processor can safely enter this mode from protected mode, run an 8086 program, and return to protected mode. All code runs at privilege level 3. Protected segmentation is not available.

The descriptions in this manual assume protected-mode operation. The section entitled "Other Processor Modes" in Chapter 4 focuses specifically on the real mode and virtual-8086 mode. Most application instructions work the same way in all three modes. The operational differences between the various modes are discussed briefly below and in detail in Chapter 4, "System Programming."

Segmentation works differently, depending on the mode. In protected mode, the segment selector is used as a pointer into a segment descriptor table. The descriptors in the table specify the base and limit of the segment in linear address space, and they enforce segment access restrictions based on privilege level. In real mode and virtual-8086 mode, the segment selector is multiplied by 16 to form the base address of the segment; each segment is therefore 64kB in size. There is no segment-level protection.

Page translation, with full page-level protection, is available in protected mode and virtual-8086 mode. Paging is not available in real mode.

The use of instructions that access I/O devices, like IN and OUT, can be restricted in protected mode and virtual-8086 mode to code of a certain privilege level (the IOPL). In virtual-8086 mode, instructions that reference the interrupt flag (IF) are also sensitive to the IOPL.

Handling or service routines for interrupts and exceptions are located using a vector into a data structure in memory. This table has two formats, one for real mode and the other for protected and virtual-8086 mode. In real mode, the table is called an *interrupt vector table*. In protected mode and virtual-8086 mode, the table is called an *interrupt descriptor table*.

Data Types

The supported data types include unsigned and signed integers, binary-coded decimal numbers, strings (including bit strings), and pointers. These types are described later. Floating-point data types are supported by numerical coprocessors that the Super386 processor in turn supports—such as the SuperMath and standard 80387 coprocessors—and by software packages that emulate coprocessors. For details on these floating-point data types, refer to the documentation for the coprocessors or emulation software.

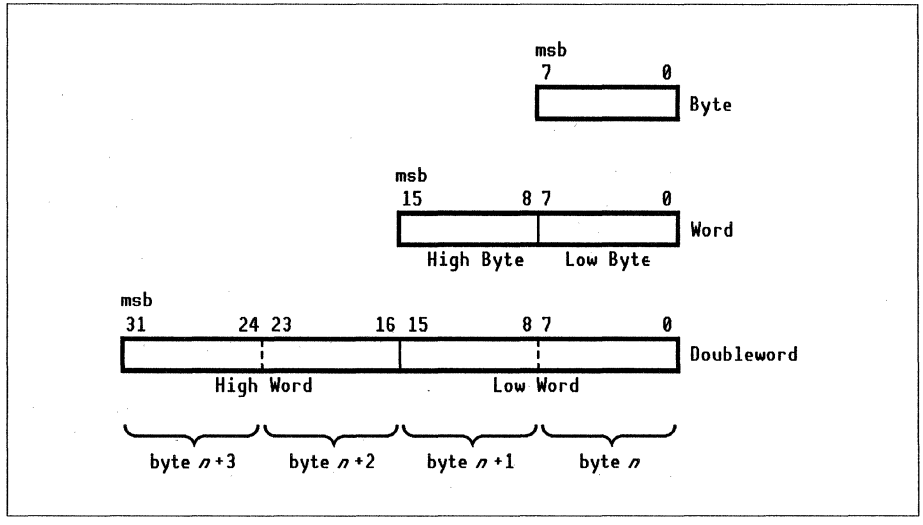
Integers

The processor supports the representation of integers in unsigned and signed formats of various widths.

Unsigned Integers

An unsigned integer represents a non-negative value in binary (radix-2) form. Unsigned numbers can be a byte, word, or dword in length, as shown in Figure 2-14. An unsigned byte can represent integers between 0 and 255 (inclusive). For unsigned words, the range is from 0 to 65,535; for unsigned dwords, from 0 to 2^{32}-1.

Figure 2-14. *Unsigned Integers*

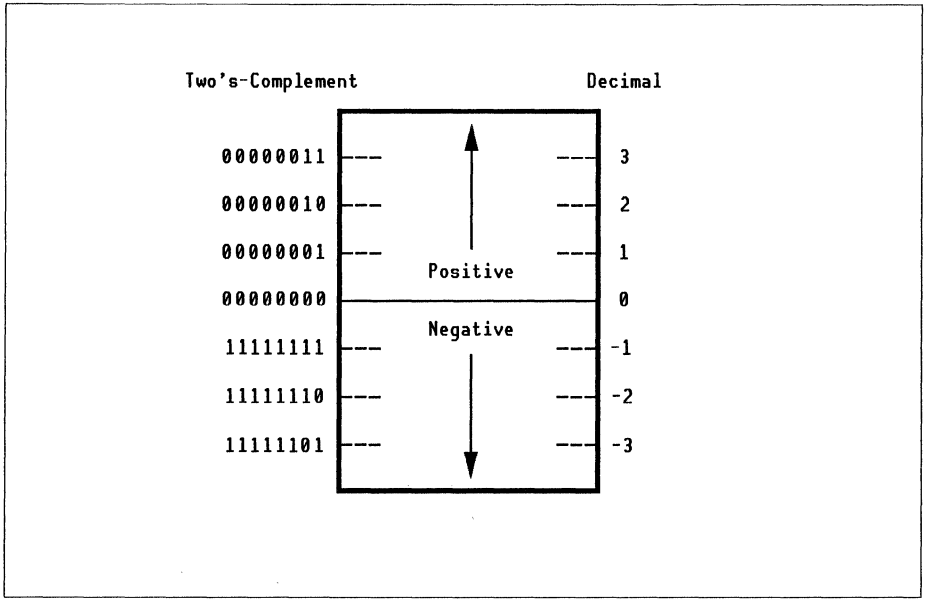


The various instructions that add or subtract integers work equally well with unsigned and signed integers. Special instructions supporting unsigned numbers are available for multiplication and division.

Signed Integers

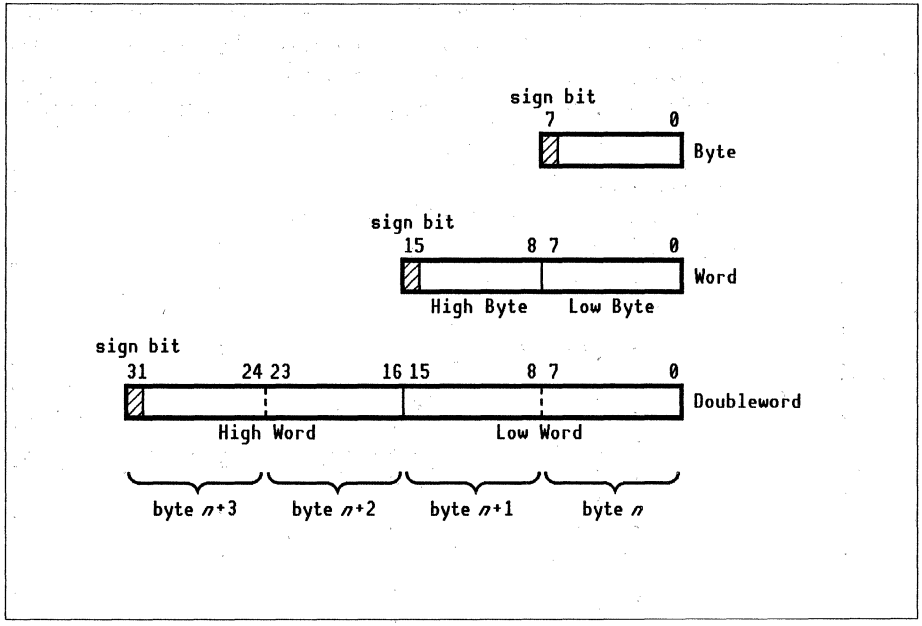
A signed number represents an integer in *two's-complement* format, as shown in Figure 2-15. In this format, the most-significant bit indicates the sign: 0 for positive, 1 for negative. The remaining bits indicate the magnitude. For positive numbers, these bits directly represent the magnitude in binary (radix-2) form. For negative numbers, every bit of the absolute value in binary form is inverted (one's complement), and 1 is added to the result.

Figure 2-15. *Two's-Complement Integers*



Signed numbers can be a byte, a word, or a dword in length. An *n*-bit signed number can represent integers between -2^{n-1} and $+2^{n-1}-1$. The signed number types are illustrated in Figure 2-16.

Figure 2-16. Signed Integers



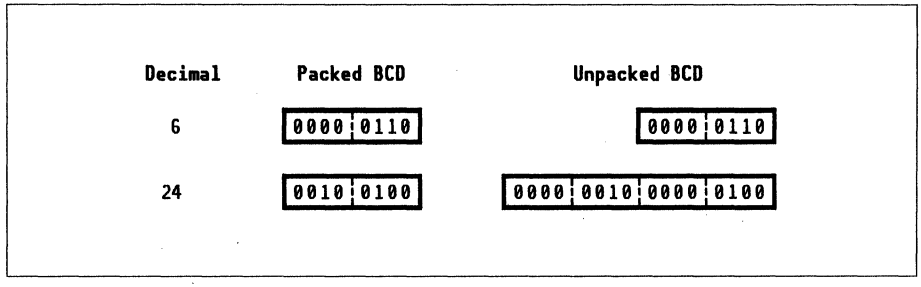
The various instructions that add or subtract integers work equally well with unsigned and signed integers. Special instructions supporting signed numbers are available for multiplication and division.

Quadword numbers (8 bytes long) also occur. They are generated by the 32-bit multiply instructions. The low-order dword is normally stored in register EAX and the high-order dword is stored in register EDX. Similarly, in a 32-bit divide instruction, the dividend is a quadword taken from the EAX and EDX registers.

Binary-Coded Decimal (BCD) Numbers

In a BCD number, each digit of a decimal numeral is represented in binary form (from 0 = 0000b to 9 = 1001b). In the *unpacked* BCD representation, each digit is stored in a separate byte. Alternatively, two digits can occupy a single byte in *packed* BCD format, where the digit represented by bits 7:4 is more significant than the digit in bits 3:0. Figure 2-17 illustrates both varieties.

Figure 2-17. *Binary-Coded Decimal Numbers*



Special BCD arithmetic instructions act directly on one-byte BCD numbers. Multi-byte BCD numbers must be handled as strings (see "Strings" on page 2-24). BCD strings do not have a set length and can therefore be used to represent numbers of arbitrary precision.

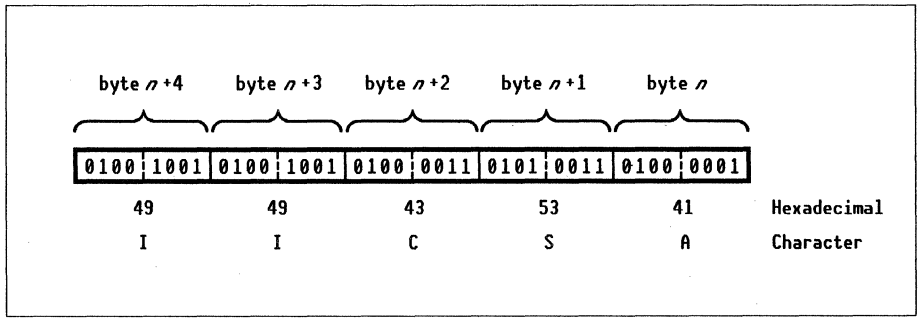
Strings

A string is a sequence of bits, bytes, words, or doublewords that occupies a single contiguous block of memory. The processor operates on a string by applying a specified string instruction to each successive element. There are instructions for moving strings around in memory, filling a string with repetitions of a fixed value, transferring strings between memory and I/O ports, and searching strings for specific values. The string instructions are discussed in Chapter 3, "Instruction Set Overview." Bit strings can contain up to 2^{32}-1 bits. Other types of strings can be up to 4GB in size.

ASCII

The American Standard Code for Information Interchange (ASCII) represents alphanumeric and control characters in a 7-bit binary code. Sequences of ASCII-encoded characters are among the most commonly used strings. Each byte of an ASCII string contains a character in bits 6:0. Bit 7 is cleared to 0. The processor can perform arithmetic operations on one-byte ASCII code numbers. Figure 2-18 shows an ASCII string.

Figure 2-18. ASCII String

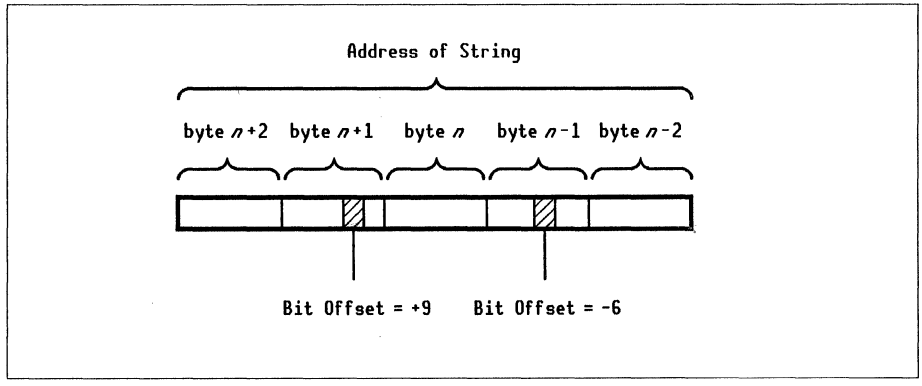


Bit Strings

Bit operations support data that does not break down conveniently into bytes, such as a display bitmap or a single-bit datum like a semaphore. In the former case, it would be inconvenient to have to manipulate the data in byte-sized pieces. In the latter case, it would waste memory to use an entire byte in order to store a single bit.

Bit strings are indexed by a dword, and can therefore be up to 2^{32} bits in length. The index is a signed integer called the *bit offset*. It specifies the location of a specific bit within the string. Figure 2-19 gives an example of bit addressing.

Figure 2-19. *Addressing a Specific Bit*

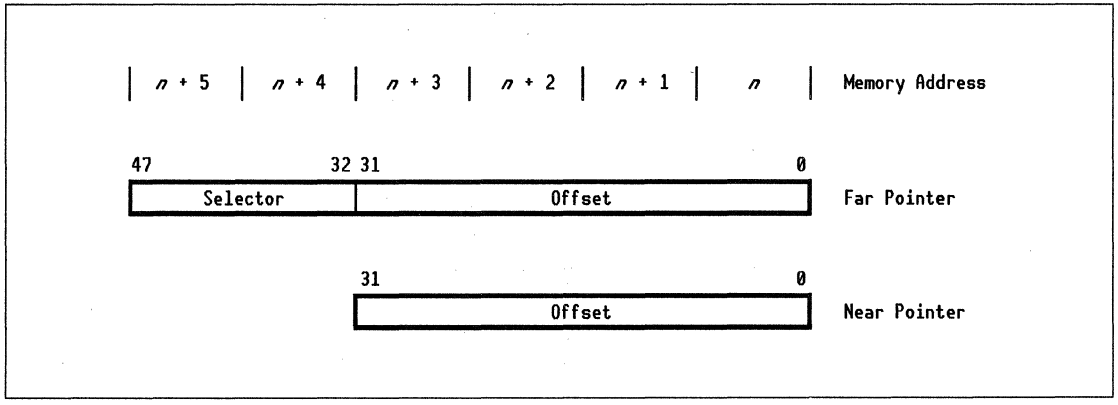


Pointers

A pointer contains the address of a data item. Pointers can be used to build and access complex data structures that can change in size and structure during execution. Each element in a linked list, for example, contains a pointer to another element. Elements can be linked and unlinked by writing new values to the pointers.

There are two types of pointers, a *far pointer* containing a segment selector as well as an offset, and a *near pointer* containing just an offset. See Figure 2-20.

Figure 2-20. *Near and Far Pointers*



Far Pointer—A far pointer contains a two-part address which is required for accessing an element located in a different segment of memory. The offset part is stored in the low-order 32 bits (16 bits in real and virtual-8086 modes), and the segment selector is in the high-order 16 bits.

Near Pointer—A near pointer contains only an offset. Near pointers can only be used when all pointer references lie in one segment.

Instructions exist for loading pointers from memory. The segment selector (for far pointers) is loaded into a segment register. The offset is loaded into a general register, to be used as the base in an address calculation.

Application Registers

The 16 registers available to application programs are shown in Figure 2-21. Application registers are of three kinds:

- General registers
- Status and control registers
- Segment registers.

The registers are discussed briefly in the following paragraphs. Full details are given in Appendix A, "Instruction Set Reference."

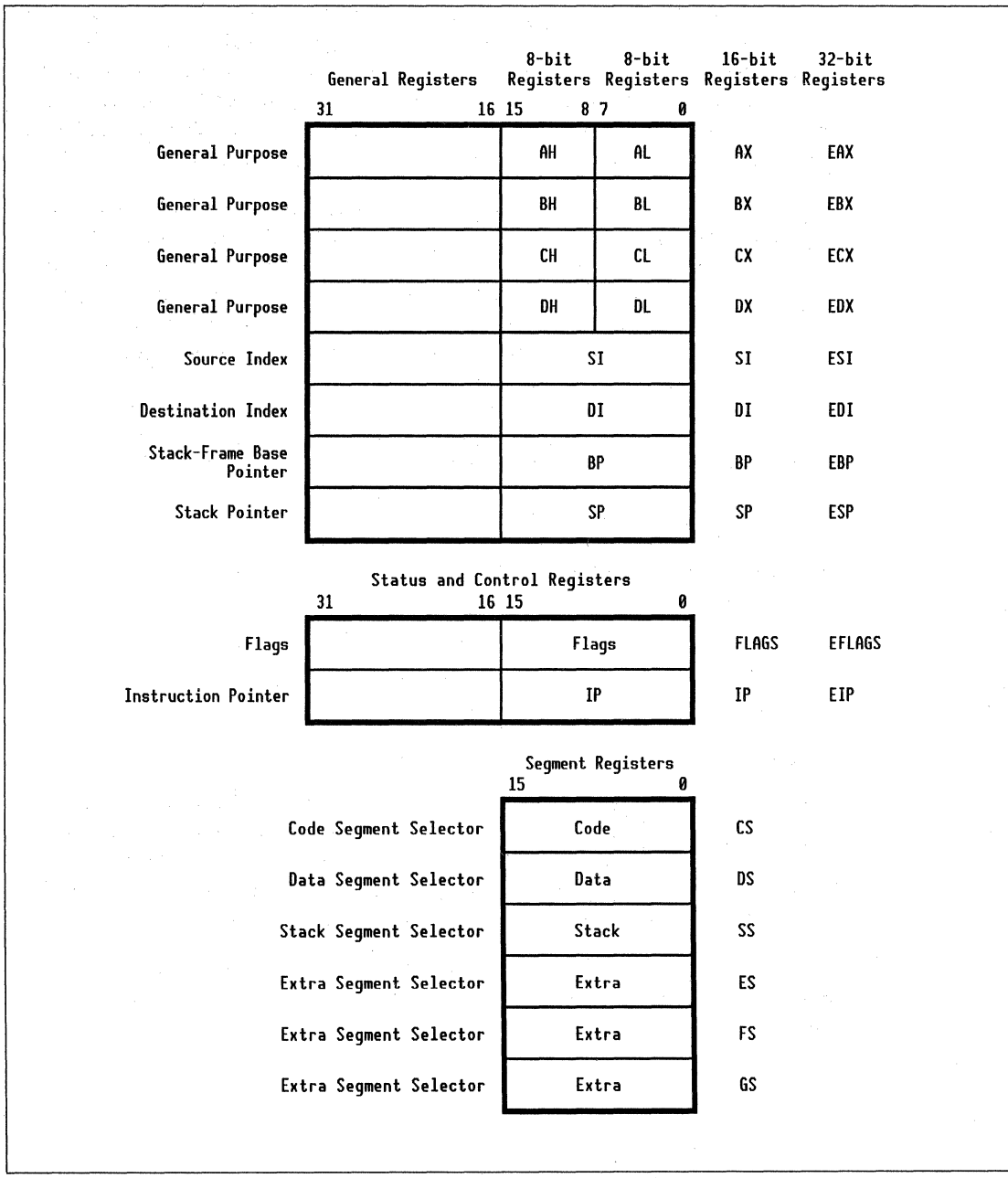
General Registers—The eight 32-bit general registers, also called *general purpose registers,* are EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP. They are used for a variety of programming operations, such as holding intermediate results in computations, holding base and index values for address computations, and holding parameters and local variables during subroutine calls. Some instructions use one or more of the general registers in a special way.

Status and Control Registers—Status and control registers, EFLAGS and EIP, are 32-bit *flag* registers. EFLAGS contains bits that either modify the effect of instruction execution, reflect the outcome of instruction execution, or configure certain system-level resources. The 32-bit *instruction pointer* (EIP) register acts as the program counter.

Segment Registers—The six 16-bit segment registers (CS, DS, SS, ES, FS, and GS) contain selectors that identify the currently addressable code segment (CS), data segment (DS), or stack segment (SS) of memory. The ES, FS, and GS are extra data-type segments.

The general registers plus the status and control registers are sometimes called the *base register set* in other literature. In addition to these application registers, the processor also has several other registers—usually not available to application programs—through which segmentation, paging, debugging, testing, and other system operations are controlled. The registers typically used by application programs are described below. Those typically used by system programs are described in Chapter 4, "System Programming."

Figure 2-21. Registers Available to Application Programs



General Registers

The eight general registers shown in Figure 2-21 support doubleword, word, and byte operands. The full 32-bit registers have names that begin with E (for *extended*). To handle 16-bit operands, the lower word of each general register is separately addressable. It has the same name as the full 32-bit register, minus the E. Four of the general registers (those whose names end in X) also support 8-bit operands. In these registers, each byte of the lower word is separately addressable. The high-order bytes are AH through DH. The low-order bytes are AL through DL.

Some instructions operate on bytes, others on words or dwords. Those that operate on words or dwords determine the operand size via a bit (the default size) in the segment descriptor for the code segment. An instruction prefix called the *operand-size override* allows switching between operand sizes. Byte and word operations that modify a general register affect only the specified portion of that register. The other bits remain unchanged. When a general register is pushed on or popped from the stack, the operand size matches the operand; the remaining bits are undefined (see PUSH).

Most instructions can use any of the general registers as operands. Some instructions, however, implicitly use one or more of the general registers in a special way:

- String instructions
- Double-precision arithmetic
- Variable shifts
- Input/output instructions
- Stack manipulation.

These uses are discussed in the following paragraphs.

String Instructions—Strings are processed by applying a specified instruction to each element in a string. The source index (ESI) register and destination index (EDI) register indicate the operation's source and destination strings. These registers are incremented or decremented as each successive element of the string is processed. The value in ECX is interpreted as the total length of the string.

Double Precision Arithmetic—EAX and EDX together hold the 64-bit product in a double-precision multiplication. They hold the 64-bit dividend in a double-precision division.

Variable Shifts—For some shift instructions, the CL register specifies the number of bits to be shifted.

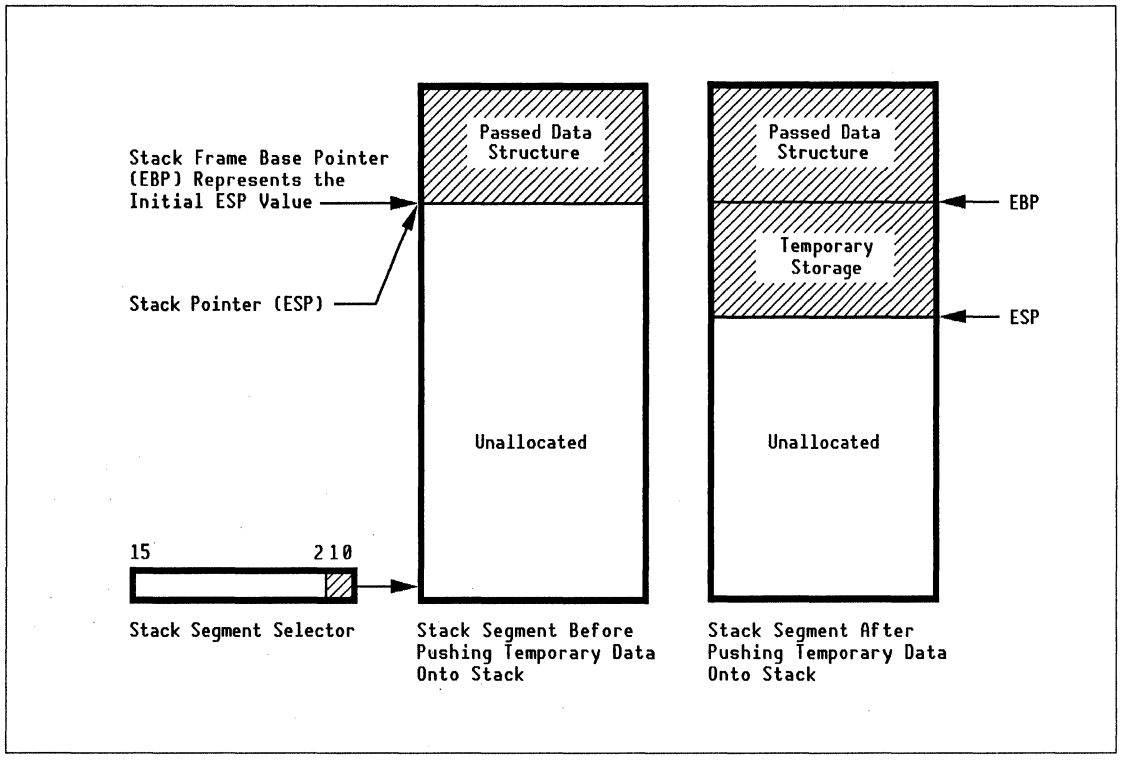
Input/Output—I/O instructions use the EAX, AX, and AL registers as sources for output data and as destinations to receive input data. Block I/O transfers use the DX register to specify a port in I/O space, and they use the source and destination index registers ESI and EDI as string indexes.

Stack Manipulation—The stack pointer (ESP) and stack-frame base pointer (EBP) registers are used for stack manipulation. They contain offsets into the current stack segment. The ESP register contains the offset of the current top of stack. It is decremented when an item is added to the stack and incremented when an item is removed. The stack thus grows down toward lower memory addresses. Figure 2-11 illustrates the stack storage-allocation discipline.

The entire structure of the stack, including the stack pointer and its stack-frame base pointer, is called the *stack frame*. The base pointer in the EBP register is typically used as a fixed reference point for accessing the stack in situations where the stack pointer itself is changing. For example, suppose a data structure is passed on the stack to a subroutine that also uses the stack for temporary storage of local variables, as shown in Figure 2-22. In this situation, ESP-relative addresses for data in the fixed data structure would have to change as the amount of temporary storage allocated for local variables changed. By copying the initial ESP value into the EBP register before pushing anything onto the stack, the subroutine can instead use fixed, EBP-relative addresses to access the passed data structure.

The ENTER and LEAVE instructions automatically set up a stack frame for procedures and exit from them. The instruction descriptions in Appendix A, "Instruction Set Reference," give full details concerning these instructions and all of the implicit uses of general registers.

Figure 2-22. *Use of the Stack-Frame Pointer*



Status and Control Registers

The two status and control registers, illustrated in Figure 2-21, are of significance to application programmers. One points to the current or next instruction, and the other contains control and status flags.

Instruction Pointer (EIP) Register

The EIP register contains an offset into the current code segment, which is the segment pointed to by the value in the CS register. The EIP register is loaded automatically by an interrupt, an exception, or a control-transfer instruction such as JMP or RET. For 16-bit addressing, the lower word (IP) of the EIP register provides the offset. When used independently, this portion of the EIP register is called the IP register.

Status and Control Flags (EFLAGS)

The EFLAGS register, shown in Figure 2-23, has 13 non-reserved flag fields. There are three kinds of flags:

- Status flags
- Control flags
- System flags.

These flags are discussed in the following paragraphs.

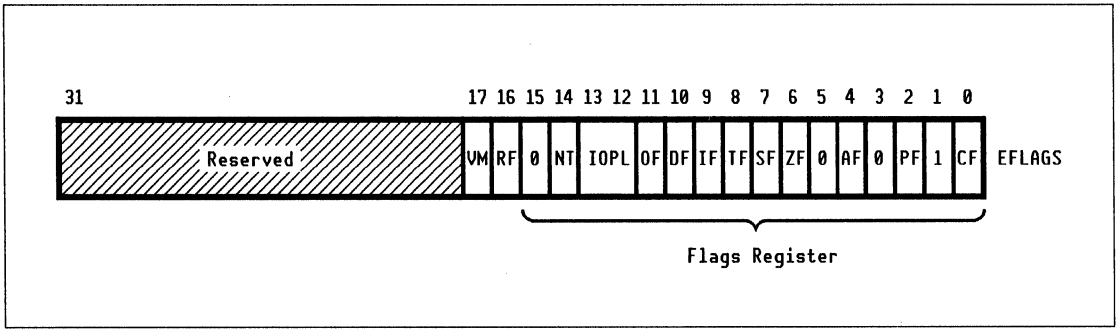
Status Flags—Status flags provide information concerning the result of the last arithmetic instruction to be executed. The status flags show whether the result was positive, negative, or zero; whether overflow occurred; and other similar conditions. Conditional jumps and software interrupt INTO calls read and respond to these flags.

Control Flag—The DF flag is the only control flag. It controls the direction of string operations. The direction flag can be explicitly set or cleared.

System Flags—There are several system flags for controlling I/O, interrupts, debugging, multitasking, and operating mode. These flags are described in Chapter 4, "System Programming."

Only the status and control flags are described. The effect of each instruction on the flags is specified in Appendix A, "Instruction Set Reference."

Figure 2-23. EFLAGS Register



bits: 11 OF — Overflow Flag—Indicates whether the uppermost bit (sign bit) of an operand is changed as a result of an operation. After a shift operation of more than one bit, the OF flag is undefined:
1 Overflow
0 No overflow

10 DF — Direction Flag—Indicates whether a source or destination address pointer of a string instruction (the contents of the ESI and/or EDI register) should be incremented or decremented after each iteration of the instruction execution. The flag can be explicitly set or cleared by the STD and CLD instructions.
1 Decrement
0 Increment

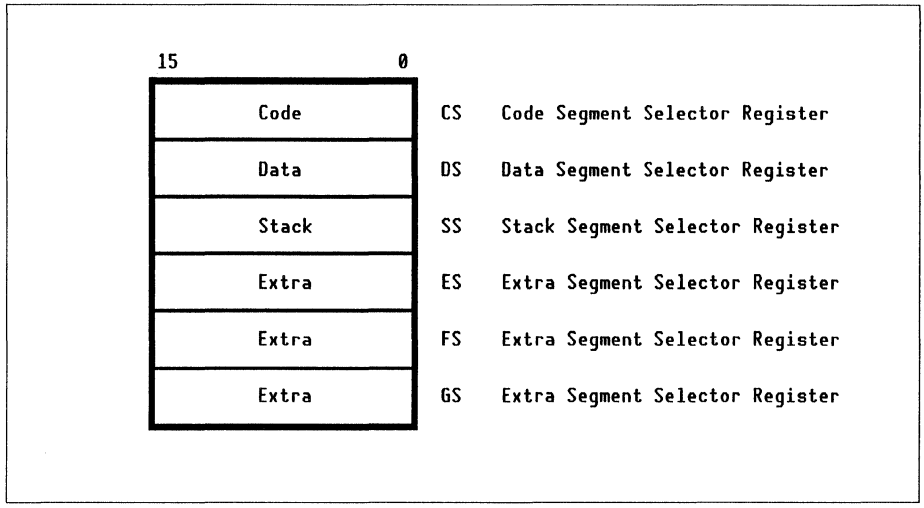
7 SF — Sign Flag—Indicates whether an arithmetic operation had a positive or negative result, as indicated by the high-order bit of a byte, word, or doubleword (bit 7, 15, or 31):
1 Negative result
0 Positive result

6	ZF	Zero Flag—Indicates whether an arithmetic operation resulted in zero: 1 Zero result 0 Nonzero result
4	AF	Auxiliary Flag—Indicates whether a BCD arithmetic operation resulted in a carry out (addition) or borrow (subtraction) from bit 3 of the least-significant byte, regardless of the operand size: 1 BCD carry out or a borrow occurred 0 No BCD carry out or a borrow occurred
2	PF	Parity Flag—Indicates the number of 1s in the low-order operand byte after an arithmetic operation, regardless of the operand size: 1 Even number of 1s 0 Odd number of 1s
0	CF	Carry Flag—Indicates whether an arithmetic operation resulted in a carry out (addition) or borrow (subtraction) into the high-order bit: bits 6, 14, and 30 for signed integers; bits 7, 15, and 31, respectively, for unsigned integers. The flag can be explicitly set or cleared by the STC and CLC instructions. The flag can be complemented with the CMC instruction: 1 Carry out or borrow occurred 0 No carry out or borrow occurred

Segment Registers

There are six 16-bit segment registers available to application programs. Each segment register contains the selector for one memory segment. The registers are illustrated in Figure 2-24 and listed below.

Figure 2-24. *Segment Register*



CS Code Segment—References the currently active executable code segment.

DS Data Segment—References the currently active data segment.

SS Stack Segment—References the currently active stack segment. The SS register can be loaded explicitly, allowing application programs to set up stacks. There can be as many stacks as the number of segments.

ES Extra Segment—References the currently active segment that must be used to hold the destination operands for string instructions.

FS Same as ES—Extra data segment.

GS Same as ES—Extra data segment. This register is also used as an override prefix to access user memory in SuperState V mode on the 38605 processor.

The six segments can be directly addressed by the processor. To access a segment, the processor loads its selector into one of the segment registers. In real mode, the selector is multiplied by 16 to locate the base address of the corresponding segment. In protected mode, the selector points to a segment descriptor contained in a memory-resident table. The descriptor contains information about the segment, including its base address and its size limit. For the segments whose selectors are currently in the segment registers, the descriptor information is automatically cached on-chip in registers not directly accessible to software.

The processor fetches instructions from the segment located by the selector in the CS register. The CS register cannot be explicitly loaded by software. Instead, its value can be changed only by executing a far control transfer, in which a CALL or JMP instruction references a code segment other than the current code segment.

The selector for the current stack segment is in the SS register. All stack operations access this segment. Unlike the CS register, the SS register can be loaded explicitly. This feature allows application programs to set up stacks. The DS, ES, FS, and GS registers hold selectors for data segments. Application programs can also load values into these registers. When a selector has been loaded into its appropriate register, an instruction needs only to provide an offset for the processor to form a complete logical address.

Interrupts and Exceptions

Interrupts and exceptions are responses to exceptional events. The processor temporarily suspends the flow of normal program execution, transferring control to a handling routine that services the event and returns control to the suspended program.

Interrupts

Interrupts occur in one of two ways:

- System hardware requests the attention of the processor by asserting a signal on one of the interrupt input pins.
- Software requests an interrupt by means of an INT, INTO, or BOUND instruction.

Hardware-initiated interrupts occur asynchronously with respect to instruction execution. Interrupts are serviced either when the currently executing instruction is completed, or, in the case of instructions that could conceivably run for a long time,

when the instruction comes to a well-defined stopping point. String instructions, for example, are interruptible between operations on successive string elements.

Applications can request service from an operating system interrupt handler by using the INT *n* instruction, where *n* is the interrupt (or exception) vector. However, interrupt vectors that do not correspond to an interrupt handler defined by the operating system must be handled by the application program itself.

Exceptions

Exceptions are the result of abnormal conditions detected during the course of instruction decoding or execution. For example, exceptions occur when instructions are improperly coded, violate protection rules, or access pages that are not present in memory. Exceptions and software-initiated interrupts occur synchronously with respect to instruction execution. Exceptions differ from one another in the state of the machine upon entering the service routine. There are three types: faults, traps, and aborts.

Faults—A fault occurs when the instruction that caused the exception is nullified; i.e., the machine state prior to that instruction is restored before the fault handler is invoked. The instruction is typically retried after the fault condition is repaired.

Traps—A trap results when the instruction that caused the exception, but no other instruction, is completed before the trap handler is invoked. Software interrupts can be considered traps. Certain breakpoint exceptions used in debugging are also traps.

Aborts—In an abort, the instruction that caused the exception, and possibly several others as well, complete before the handler is invoked. Or, an exception is reported while another exception is being processed. If a double-fault abort is followed by another exception, the processor will shut down and require a reset.

Page faults are common examples of fault exceptions. A page fault occurs when an instruction accesses a page that does not currently reside in memory. The fault handler swaps the required page into memory, updates the page-translation tables as needed, and then retries the faulting instruction. Page faults are a normal occurrence in a demand-paged system.

Like interrupts, most exceptions are handled by the operating system. However, those that result from erroneous code or that are directly requested by an application program must be handled by the application program itself. For example, a supervisory program receiving a divide exception caused by an application will probably be unable to do anything but terminate the application. Details of interrupt and exception handling are discussed in Chapter 4, "System Programming."

On-Chip Instruction Cache

Instruction reads (fetches) from memory can be a bottleneck on the processor bus. To minimize this, a 512-byte on-chip instruction cache is provided in the 38605 processor. Instruction prefetching into this cache reduces the effect of external memory latency on prefetching, and reduces interference with operand accesses, thus improving the processor's performance. The cache works in conjunction with a 12-byte instruction buffer that can accept instructions from the instruction cache at the rate of four bytes per cycle. With the cache enabled, the processor fetches instructions from memory only when the next instruction is not in the cache.

Jump instructions can be executed in two clocks if the destination instruction is in the cache. By comparison, it requires five clocks to execute the jump instruction if it is not in the cache, has a prefix, or does not have an 8-bit displacement.

To take full advantage of the instruction cache, the programmer should write assembly language critical routines no longer than 512 sequential bytes, starting at an address that is a multiple of 16. If a program sequence fits entirely in the cache, sequential and near-jump instruction fetches will not interfere with operand accessing. As a rule, a routine of up to approximately 150 assembly language instructions can fit in the cache in protected mode (up to about 200 instructions in real mode).

CHAPTER 3

Instruction Set Overview

The Super386 instruction set is a superset of the Intel® 80386 instruction set. It includes a wide variety of arithmetic, logical, data-movement and control-transfer operations. These operations can be performed using data in registers, memory, or I/O space, or data that is encoded as an immediate operand in the instruction itself.

Most instructions can be used in application programs. Instructions dedicated to the protection features of the processor, however, can only be used in system programs with the appropriate privilege level. Instructions that access I/O space can be restricted by the operating system on the basis of both privilege level and the specific I/O port that is addressed.

Some instructions are restricted to operands of a particular type, or to data contained in a particular register. An effective assembly language programmer should be aware of these limitations.

This chapter discusses the basic instruction format, operand types, addressing modes, flags, and condition codes. It gives an overview of the instruction set, grouped by function, and provides guidelines for using the instructions efficiently. Appendix A, "Instruction Set Reference," contains the details of instruction encoding. This appendex lists the instructions alphabetically by assembler mnemonic and provides detailed information on the behavior of each instruction. Appendix B, "Super386 Quick Reference," contains an opcode summary.

Basic Instruction Format

An instruction specifies an operation to be performed, the location or value of the source data to be used (if any), and the location where the result (if any) should be stored. Figure 3-1 illustrates the syntax for constructing instructions from a menu of parts. The entire instruction cannot exceed 15 bytes in length.

Figure 3-1. Basic Instruction Format

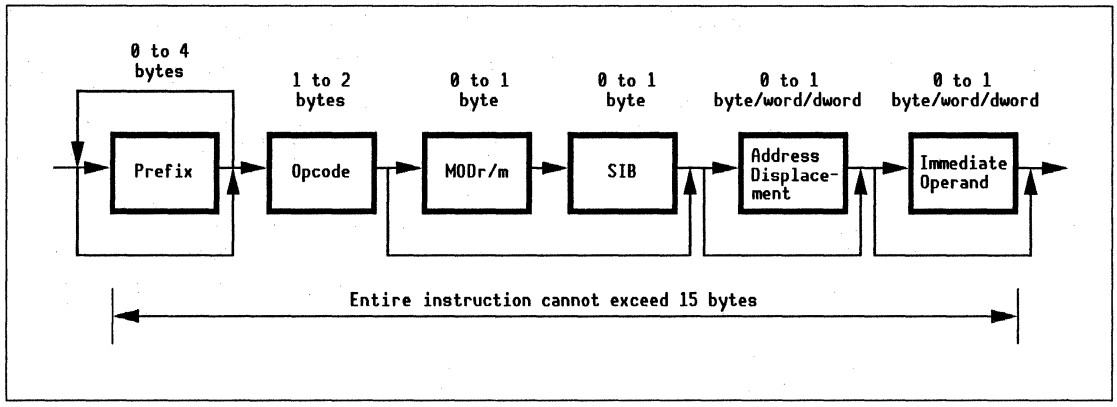


Table 3-1 describes the parts of an instruction in the order of their appearance in the instruction.

Table 3-1. The Parts of an Instruction

Instruction Part	Size	Number Required	Comments and Restrictions
Prefix	byte	0 to 4	
Opcode	byte	1 or 2	If the opcode is two bytes long, the first byte is 0Fh.
MODr/m	byte	0 or 1	Also spelled ModR/M, MOD/RM, and MODRM. Encodes a variety of attributes about source and destination, displacement, addressing mode, and instruction function. See Appendix B for the encoding.
SIB	byte	0 or 1	Specifies scale, index, and base in certain 32-bit addressing modes. See Appendix B for the encoding.
Address displacement	byte, word, or dword	0 or 1	A constant value that is added to the base and index of a decoded address to generate the effective address.
Immediate operand	byte, word, or dword	0 or 1	The name is sometimes shortened to "immediate."

Prefixes

A prefix overrides the defaults or behavior of the instruction that follows. It has no effect on subsequent instructions. There are five types of prefixes:

- Segment
- Operand size
- Address size
- Lock
- Repeat.

A segment prefix changes the default data segment for a memory operand.

An operand size prefix changes the default operand size specified in the descriptor for the current code segment. Operands can be either 8 bits or 16 bits, or they can be 8 bits or 32 bits. See the section entitled "Operand Sizes."

An address size prefix changes the default address offset specified in the descriptor for the current code segment. Address offsets can be 16 or 32 bits wide.

The lock prefix causes a memory read/modify/write operation to be performed indivisibly, as in updating a semaphore.

A repeat prefix is used with a string instruction to apply the instruction sequentially to each element in a string. Up to two repeats can be used.

Appendix B includes a reference list of all prefix values.

Opcode

The operation code, or opcode, determines the operation to be performed and, in many cases, the type of operands to be used. Every instruction has an opcode. Some instructions consist solely of an opcode.

Because of the complexity of instruction encoding, many identical operations have multiple encodings. For example, adding an immediate value to register AL can be done with a direct form (ADD AL, imm) in which the destination register AL is implied by the opcode, or with a longer form (ADD reg, imm) in which the destination register AL is explicitly coded. In such cases, the shorter form reduces code size.

Appendix B includes a reference list of all opcodes.

MODr/m Encoding

Many instructions include a MODr/m byte following the opcode. This byte is used either to determine two operands, or one operand and the operation to take place. The MODr/m byte is divided into three fields:

- MOD (mode)
- REG (register)
- r/m (register/memory).

These fields are discussed below.

MOD—The most-significant two bits form the MOD field. This field determines whether the operand is a register or a memory location and how large a displacement, if any, is present.

REG—The next two bits form the REG field. In one form of MODr/m encoding, this field determines a second operand. In the other form of MODr/m encoding, the REG field determines the operation to be performed by the instruction. Instructions that take the latter form are considered *group*, or *eleven-bit* opcodes. The three bits of the MODr/m field participate with the eight bits of the opcode to determine the instructions operation.

r/m—The least-significant three bits form the r/m field. These bits determine the addressing mode when the operand is a memory location (as indicated by the MOD field). When the operand is a register, the r/m field specifies the register.

The interpretation of the MODr/m byte is also affected by the address size of the instruction. Different encodings are used for 16-bit and 32-bit addresses. The details of MODr/m encoding, for instructions which include this byte, are given in Appendix B, Tables B-8 and B-9.

SIB Encoding

A MODr/m byte that specifies a 32-bit address for an operand may be followed by an SIB byte to allow greater address generation flexibility. If this occurs, the MOD and r/m fields determine only the size of the displacement, if any. The SIB byte determines both the index and base registers for address generation. In addition, a selectable amount of scaling can be applied to the index register. The three parts of the SIB byte that control these functions are the 2-bit *scale*, 3-bit *index*, and 3-bit *base* fields. The details of SIB encoding are given in Appendix B, Table B-10.

Address Displacement

Displacements are used for address generation. They provide a constant value that is added to the base and/or index portions of an address. If present, the displacement can be either one, two, or four bytes in size. One-byte displacements are extended to the size of the generated address by extending their sign bit.

Immediate Operand

Immediate operands are constants contained within the instruction itself. They may be one, two, or four bytes in size, depending on the opcode and the operand size. In some cases, the REG field of the MODr/m byte determines the presence of an immediate operand. One-byte immediate operands are sign-extended to the size of register or memory operands; that is, the value of their sign bit (the highest-order bit in the byte) is used to fill the additional bit positions in the larger operand.

Operands

Most instructions require one or more operands that specify data values or locations to be used by the instruction. Operands are either explicitly encoded in a field within the instruction or are implied by the opcode. A *source operand* specifies a data value or location that is used, but not modified, by the instruction. A *destination operand* specifies a location whose value is changed by the instruction. An operand belongs to one of four types, depending on its location:

- Register
- Memory
- I/O port
- Immediate.

The operand types are discussed below.

Register—The register operands include the general registers, control registers, debug registers, test registers, flag register, and instruction pointer. These registers are described in Chapter 2, "Programmer's Model," and Chapter 4, "System Programming."

Memory—These operands specify a memory address as source or destination. The various addressing modes for memory operands are discussed in the section entitled "Memory Operands" in this chapter.

I/O Port—The I/O port operands reference a seperate space (different from memory addresses or registers) in which I/O devices are located.

Immediate—These operands are constant values contained within the instructions. An instruction can have multiple register operands and multiple memory operands, but only one immediate operand.

Operand Sizes

Operand sizes are implied by the instruction and are further controlled by the processor's execution mode. Table 3-2 lists the possible operand types and sizes. For many instructions, one of two operand sizes is determined by the opcode. The shorter size is always one byte; the larger size is either two bytes or four bytes, depending on the processor's execution mode.

In real mode and virtual-8086 mode, the default for the larger size is two bytes. In protected mode the default for the larger size is determined by the default (D) bit (bit 22) in the upper dword of the code segment descriptor. If D = 0, the long operand size is two bytes; if D = 1, the long operand size is four bytes. For a diagram of the code segment descriptor, see the section entitled "Segmentation" in Chapter 4, "System Programming."

The operand size instruction prefix can switch to the non-default operand size. For example, if the D bit indicates a long operand of four bytes, preceding the instruction with the operand size prefix will cause it to access a two-byte quantity. Conversely, if the D bit indicates a long operand of two bytes, or if the processor is in real mode or virtual-8086 mode, preceding the instruction with the prefix will cause it to access a 4-byte quantity.

Table 3-2. Operand Types and Sizes

Type		Size (bytes)
Register:	General	1, 2, or 4
	Control	2 or 4
	Segment	2
	EIP	2 or 4
	EFLAGS	2 or 4
	Debug	4
	Test	4
Memory		1, 2, or 4
Immediate		1, 2, or 4
I/O		1, 2, or 4

Register Operands

Application programmers are typically concerned only with the general registers and the EFLAGS register. Some may also have reason to use the segment registers. Most instructions operate only on those registers. The remaining registers are provided for system control and debugging.

Some general registers can be accessed with byte, word, or dword operands. Within such registers, smaller operands are a subset of the larger operand. For example, loading the EAX general register with 00000000h and then loading the AX portion of this register with 5A5Ah will result in a value of 00005A5Ah in the EAX register. It is not possible to access the SI, DI, BP, or SP registers using byte operands. If this is attempted, the DH, BH, CH, and AH registers, respectively, will be selected instead.

Memory Operands

Memory operands are located at the address generated by the instruction. If the operand is more than one byte wide, the least-significant byte is located at the generated address, and each next-significant byte is located at each next successively greater address. For many instructions, the opcode or the MODr/m byte determines whether an operand comes from memory or is provided by a register. The rules by which the memory address is generated are complex. They take into account the segment selected, the segment base address, the address size, and the components used to generate the effective address.

Segment Selection

At any time, the processor can directly access six memory segments by loading the DS, CS, SS, ES, FS and GS segment registers with selectors. The interpretation of these selectors differs between the real and protected modes of operation, but the effect in both cases is to enable access to six different regions of memory.

Selection of the segment depends on the instruction type and the addressing mode. Program code must be located in the CS segment because the processor only fetches instructions from there. The DS segment is the default segment for the operands of most instructions, with the following exceptions:

- Stack instructions must use the SS segment register.
- String instructions must use the ES register for the operand that is pointed to by the EDI register.
- Non-stack instructions that generate an address from a base located in either the ESP or EBP register must use the SS register.

Instructions can include a segment prefix to override the default segment, as described in the section entitled "Prefixes." It is not possible, however, to override the segment used for stack operands, string-destination operands, or code fetches. Any attempt to do so is ignored. There are no instructions that access the FS segment and GS segments directly; a segment prefix must be used to access them.

Address Size

The processor generates 16-bit or 32-bit addresses, depending on the setting of the default (D) bit (bit 22) in the upper dword of the code segment descriptor. In real mode, the D bit is cleared to 0, causing the default address size to be 16 bits. In protected mode, the D bit can be cleared to 0 or set to 1. If it is set, the default address size is 32 bits. The size of an address can be altered by preceding the instruction with an address-size prefix.

For control-transfer instructions, the size of the target address is determined by the D bit and the operand-size prefix, rather than the D bit and the address-size prefix. The target-address size also determines the size of the displacement field in the direct forms of control transfer.

For a diagram of the code segment descriptor, see the section entitled "Segmentation" in Chapter 4, "System Programming."

Addressing Modes

An effective address must be generated before segmentation is applied. There are six modes by which the effective address is generated: absolute, stack, instruction-relative, string, complex, and register.

Absolute Addresses

A few instructions move the contents of the AL, AX or EAX register to or from a location in memory that is pointed to by the displacement field of the instruction. The default segment is DS, and the displacement is treated as an unsigned offset into the segment. The long operand form of move can toggle between the AX and EAX registers using the operand size prefix.

Stack Addresses

Stack addresses are generated by PUSH and POP instructions, including the PUSH mem and POP mem instructions, as well as by instructions such as CALL, RET and INT. The CALL instruction generates a stack address when it pushes its return address on the stack. Similarly, the INT instruction generates stack addresses for each of the operands it pushes on the stack.

The stack address size is determined by the the big (B) bit (bit 22) in the upper dword of the stack segment descriptor. If B is 0, a 16-bit address is generated; if B is 1, a 32-bit address is generated. (For a diagram of the stack segment descriptor, see the section entitled "Segmentation" in Chapter 4, "System Programming.")

The address size prefix does not work with stack addresses, just as it is not possible to override the selection of the stack segment. The operand size prefix, however, works with stack operands when the segment's B bit is 0. Preceding a PUSH instruction executing in a 16-bit code segment with an operand size prefix causes a doubleword quantity to be placed on the stack and causes the stack pointer to be updated to point to the next doubleword address.

Instruction-Relative Addresses

Instruction-relative addresses are generated by control transfer instructions to access their target. Such instructions either contain a displacement or fetch from memory a similar value that is treated as a signed offset from the address of the instruction following the transfer. The displacement and address are added together to determine where the target instruction is located.

String Addresses

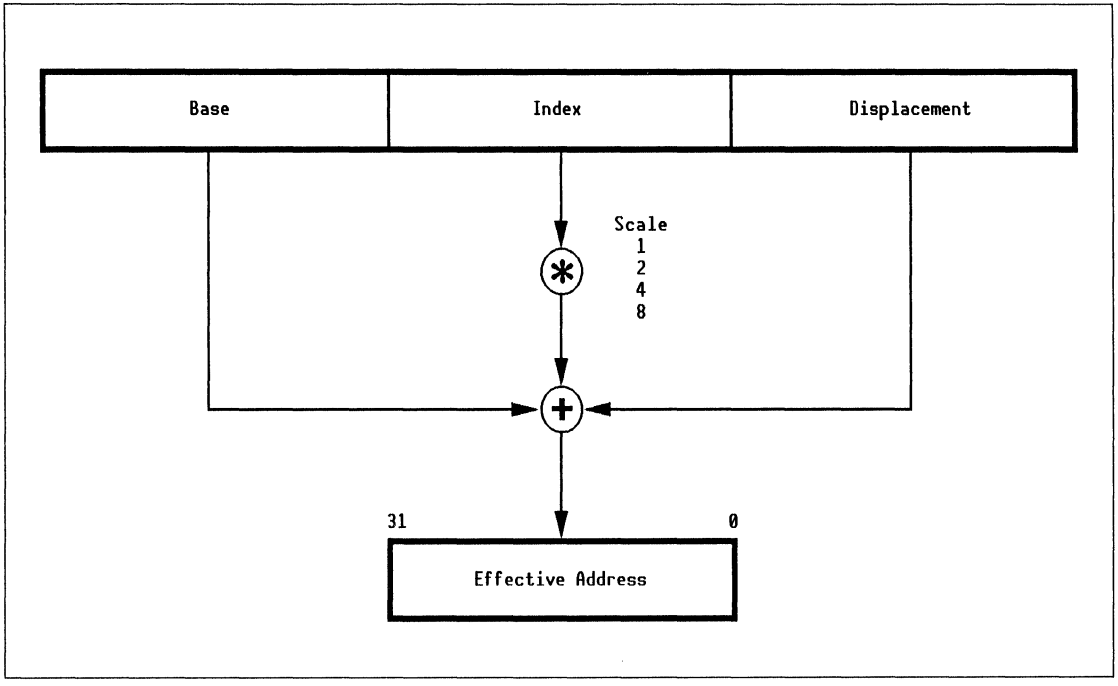
String instructions access operands in memory by generating addresses to the DS and ES segment. Each string instruction has a source and/or destination operand. The source operand is addressed by the ESI register, and the destination by the EDI register. The destination operand is always located in the ES segment. This condition cannot be overridden. Preceding a string instruction with a segment prefix will cause the source operand segment to be changed. Both the address size and operand size prefixes have their normal effect on string instructions.

Complex Addresses

The complex form of address generation is the most powerful and is available to most instructions. Those instructions that use this form always contain a MODr/m byte immediately following the opcode. When 32-bit addresses are generated, the MODr/m byte may indicate that a SIB byte follows the MODr/m byte. The MODr/m byte also indicates the presence and size of a displacement field. While interpretation of the MODr/m byte depends on the size of the address generated, the types of address generation are similar. In all cases, the segment defaults to the DS segment unless the base component is either the ESP or EBP register, in which case the SS segment is used. The MODr/m byte is also capable of selecting the operand type. It can either access an operand in memory at a complex address or it can access an operand in one of the eight general registers.

Figure 3-2 shows how 32-bit effective addresses are generated from the base address, displacement field, and index.

Figure 3-2. *Effective Address Generation*



The possible combinations of these components for generation of 8-bit, 16-bit, and 32-bit addresses are illustrated in Figures 3-3 and 3-4 and are discussed following Figure 3-4. Figure 3-3 illustrates a register view of 8-bit and 16-bit effective address generation.

Figure 3-3. *Registers Used in 8-Bit and 16-Bit Effective Address Generation*

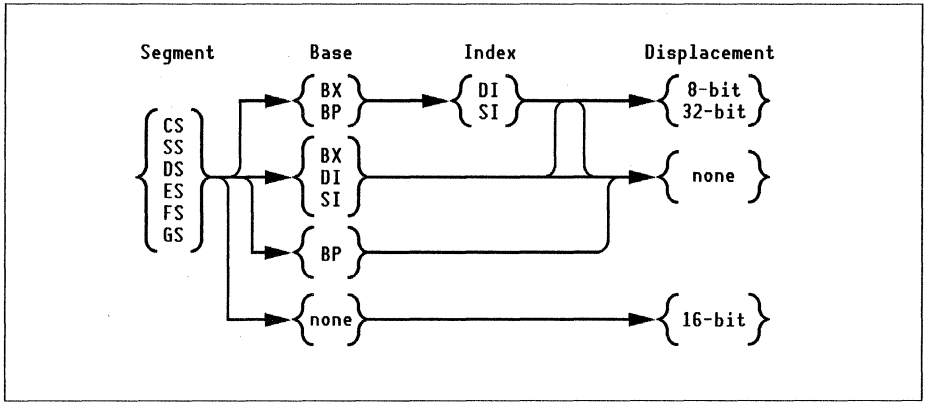
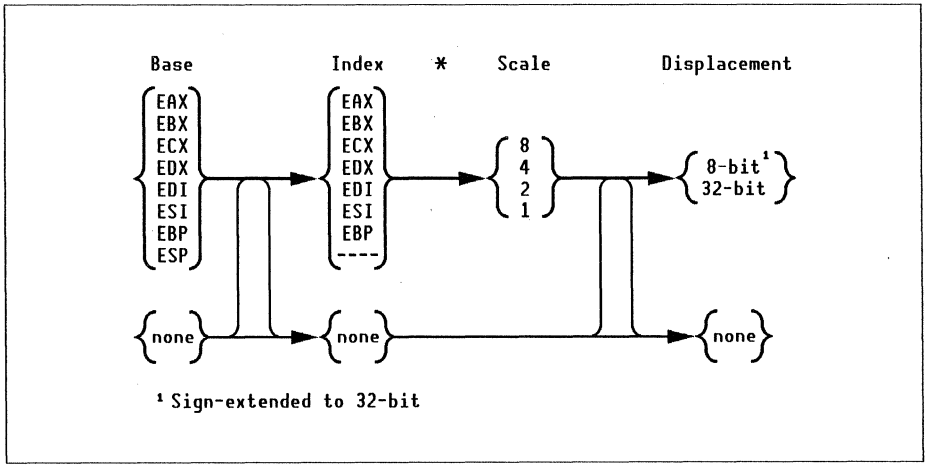


Figure 3-4 shows a register view of 32-bit effective address generation.

Figure 3-4. *Registers Used in 32-Bit Effective Address Generation*



Base Address—The base address is selected from one of the general registers. When 16-bit addresses are being generated, only the SI, DI, or BX register is used (Figure 3-3).

Displacement Field—The displacement field points to an offset within the current segment. Because no register is involved in the address generation, the segment always defaults to DS. The displacement can either be short or long. Short displacements are one byte in size. Long displacements are either two or four bytes in size, depending on the address size being generated. Displacements shorter than the generated address size are sign-extended.

Base and Displacement—When base is used with a signed displacement, 16-bit addresses can only select the SI, DI, BP or BX register as the base register, with a displacement of either one or two bytes in size (Figure 3-3). Addresses 32 bits in size have no such restriction, but they cannot select a 16-bit displacement (Figure 3-4).

Base and Index—When a base is added to an index for generation of a 16-bit address, the base can only be the BX or BP register, and the index can only be the SI or DI register (Figure 3-3). A base of BP selects the SS segment, and a base of BX selects the DS segment. Addresses of 32 bits can scale the index portion of the address (Figure 3-4). The scale operation multiplies the index by 1, 2, 4, or 8. When scaled by 1, the address is not changed. Any other scale amount allows it be interpreted as an ordinal pointer to 2, 4, or 8 byte quantities.

Base, Index and Displacement—When base, index, and displacement are combined, 16-bit addresses are restricted to using either the BX or BP registers for the base and either the SI or DI registers for the index. The displacement is either one or two bytes in size for 16-bit addresses, and 1 or 4 bytes in size for 32-bit addresses.

Index and Displacement—Index and displacement can be used only for 32-bit addresses. No base is present. Instead of a base, a scaled index is added to a one-byte or four-byte displacement. Selecting an index from the EBP register will cause the stack segment to be selected instead of the data segment.

Register Addresses

This is not a memory address, but it is selectable by the MODr/m byte. Eight MODr/m encodings are provided to select the eight general registers instead of a memory location. When this occurs, the default address size and segment are meaningless. The operand size instruction prefix can still be used to select the size of register accessed.

Immediate Operands

Immediate operands, or *immediates*, are contained within the instruction in the same order in which memory operands are stored. They can be bytes, words, or dwords. Because these operands are part of the instruction, their length directly affects the length of the instruction.

I/O Operands

The behavior of I/O ports depends on the devices connected to them, which usually makes them appear different from memory locations or registers. I/O port accesses can be one, two, or four bytes long, but an access to a one-byte operand at a specific port may not return a result that is a subset of a two-byte operand access at the same port. An understanding of the I/O devices connected to the processor is essential before I/O instructions can be used to access them.

Flags

The EFLAGS register is an implicit operand of many instructions, including the arithmetic and logical operations. For example, an ADD instruction will set the flags according to the result of the operation. Similarly, execution of some instructions, like ADC, will include the setting of the flags both as an input to the operation (a source) and an output from the operation (a destination). Refer to Appendix A, "Instruction Set Reference," for details of how flags are used and updated.

In some cases, the setting of a flag is undefined after an instruction executes. For example, the MUL instruction updates the carry flag (CF) and overflow flag (OF) that correspond to the result of the operation, but it leaves the zero flag (ZF) in an undefined state. Note that your code should not depend on the state of reserved flags, as future implementations of the Super386 architecture may use these flags for another purpose.

Certain flags can be modified directly. The carry flag (CF), direction flag (DF), and interrupt flag (IF) all have dedicated instructions to allow them to be set or cleared. Other flags can be modified with the POPF or SAHF instructions.

The term *condition codes* is sometimes used to refer to certain flags in the EFLAGS register such as the overflow, carry, zero, sign, and parity flags. These flags are used by conditional jump and byte-set instructions. The Jump if Zero (JZ) instruction examines the zero flag, and if set, jumps to its target address. If the flag is not set, the instruction completes and execution proceeds to the next instruction. Some conditions are more complex. The Set if Less Than or Equal (SETLE) instruction examines the sign, overflow, and zero flags.

If the SET or Jcc (J = jump, cc = condition code) instruction is preceded by an instruction that leaves in an undefined state any flags that are required to determine the SET or Jcc condition, the result will be unpredictable.

Instruction Set

This section gives an overview of the instruction set, organized by function. Appendix A, "Instruction Set Reference," provides an alphabetical list of all instructions and full details about the operation of each one.

Data Movement Instructions

The data movement instructions, listed in Table 3-3, transfer an operand from one place to another. The operand may be located in a register, in memory, or in the instruction as an immediate operand.

Other forms of data movement are provided by the PUSH and POP instructions. These one-byte instructions move operands between the registers and the stack, and automatically update the stack pointer. Still other instructions exchange operands in two registers, or a register operand with an operand in memory. Sign-extending instructions, like CBW and MOVS, can be used to expand the size of an operand. These instructions fill the additional bit positions in a larger operand with the value of the operand's sign bit (the highest-order bit in the original operand). The IN and OUT instructions move operands between the AL, AX, or EAX register and I/O ports.

Table 3-3. *Data Movement Instructions*

Mnemonic	Description
CBW/CWDE	Sign-extend AL to AX, or AX to EAX.
CWD/CDQ	Sign-extend AX to DX, or EAX to EDX.
MOV	Transfer data between a general register or memory, between two general registers, between a segment register and memory, or between a general register and any of a segment register, control register, debug register, or test register.
MOVSX	Sign-extend a byte to a word or dword, or sign-extend a word to a dword.
MOVZX	Zero-extend a byte to a word or dword, or zero-extend a word to a dword.
POP	Pop the 80x86 stack into a general register, segment register, or memory location.
POPA[D]	Pop the 80x86 stack into all general registers (word or dword size).
PUSH	Push a general register, segment register, or memory location onto the 80x86 stack.
PUSHA[D]	Push all general registers (word or dword size) onto the 80x86 stack.
XCHG	Exchange contents of two general registers, or contents of a general register and a memory location.

I/O Data Movement Instructions

Four instructions can read or write addresses in the I/O address space: IN, INS, OUT, and OUTS. These are listed in Table 3-4. The INS and OUTS instructions are string instructions, similar to MOVS. When used with the REP opcode prefix, they transfer the number of string elements (bytes, words, or dwords) specified in the CX register. These instructions can only be used in systems that implement a standard I/O space that is separate from the memory space. They cannot be used in systems that implement memory-mapped I/O.

Table 3-4. *I/O Data Movement Instructions*

Mnemonic	Description
IN	Reads an I/O port into the AL, AX, or EAX register, depending on whether the operand size is byte, word, or doubleword. The address can be an 8-bit immediate or the contents of the DX register.
OUT	Writes the AL, AX, or EAX register to an I/O port. The address can be an 8-bit immediate or the contents of the DX register.
INS	Reads from a port addressed by the DX register to the memory space addressed by a pointer in ES:EDI (the EDI register in the ES data segment). The EDI register is then incremented or decremented by 1, 2, or 4, depending on the operand size. The DF flag in the EFLAGS register selects incrementing or decrementing.
OUTS	Writes from the memory space addressed by a pointer in ES:EDI to a port addressed by the DX register. The EDI register is then incremented or decremented by 1, 2, or 4. The DF flag selects incrementing or decrementing.

Arithmetic Instructions

Arithmetic instructions, listed in Table 3-5, include addition, subtraction, multiplication, and division. Because the addition and subtraction operations are identical on unsigned and signed numbers, only one form of instruction is required for each. Other arithmetic instructions, such as multiply and divide, require unique opcodes for each operand encoding. IMUL is used for signed operands, and MUL is used for unsigned operands. The SBB and ADC instructions are provided for cascading subtractions or additions to achieve larger operand sizes. Two 64-bit quantities can be added by first performing an ADD on the lower 32 bits, which will set the carry if the result is too large for the destination. An ADC on the upper 32 bits will then include this carry in the addition.

Dedicated instructions are provided for packed and unpacked BCD data. The use of these instructions is restrictive, and specific programming practices must be followed.

Table 3-5. *Arithmetic Instructions*

Mnemonic	Description
AAA	ASCII adjust after add (unpacked BCD).
AAD	ASCII adjust before divide (unpacked BCD).
AAM	ASCII adjust after multiply (unpacked BCD).
AAS	ASCII adjust after subtract (unpacked BCD).
ADC	Add source operand and CF to destination.
ADD	Add source operand to destination.
CMP	Compare two operands and set flags.
DAA	Decimal adjust after add (unpacked BCD).
DAS	Decimal adjust after subtract (unpacked BCD).
DEC	Decrement destination operand by 1.
DIV	Unsigned divide.
IDIV	Signed divide.
IMUL	Signed multiply.
INC	Increment destination operand by 1.
MUL	Unsigned multiply.
NEG	Compute two's complement of destination..
SBB	Subtract source operand and CF from destination.
SUB	Subtract source operand from destination.

Binary arithmetic instructions update the flags, shown in Table 3-6, to indicate details of the result. These flags are tested by conditional instructions, such as Jcc and SETcc.

Table 3-6. *Binary Instruction Flag Setting*

Flag	Description
CF	Set for 8-bit ADD where the sum of the operands exceeds 255; set for carry (AAA, ADC, ADD, DAA) or borrow (AAS, CMP, NEG, SBB, SUB) with an unsigned integer.
OF	Set if the sign of the result changes due to an arithmetic instruction on signed integers.
SF	Set if the result of an arithmetic instruction is negative.
ZF	Signed and unsigned integer; set when all bits of the result are clear.

Logical Instructions

Logical instructions operate on one, two, or four-byte quantities. Each logical operation performs a function on each bit position, independently of the other bit positions. This differs from addition, for example, where one bit can propagate a carry to the next bit.

There are five logical operations: AND, OR, XOR, TEST, and NOT. The TEST instruction is identical to the AND instruction except that only the flags are altered. Logical instructions are defined in Table 3-7.

Table 3-7. *Logical Instructions*

Mnemonic	Description
AND	Bitwise-AND source operand into destination
NEG	Two's complement negation
NOT	Bitwise-negate destination
OR	Bitwise-OR source operand into destination
TEST	Bitwise-AND two source operands and set flags
XOR	Bitwise-XOR source operand into destination

Shift and Rotate Instructions

The shift and rotate instructions (Table 3-8) alter one, two, four, or eight-byte operands by shifting or rotating them left or right by a selected number of bit positions. An additional operation, rotate with carry, adds the carry to the length of the operand. This rotates 9, 17 or 32-bit quantities.

The rotate value must be specified by a byte-long immediate in the instruction or by the register CL, or it must be implied by the opcode to be 1.

Table 3-8. *Shift and Rotate Instructions*

Mnemonic	Description	Mnemonic	Description
RCL	Rotate left through carry flag CF	SHL	Shift left arithmetic
RCR	Rotate right through carry flag CF	SAR	Shift right arithmetic
ROL	Rotate left	SHLD	Shift left logical double (funnel shift)
ROR	Rotate right	SHR	Shift right logical
		SHRD	Shift right logical double (funnel shift)

Bit Manipulation Instructions

Bit manipulation instructions (Table 3-9) allow access to a single bit anywhere within a register or a variable length field in memory. The bit location can be specified either by a register or an immediate value. Two instructions, BSF and BSR, are provided so that the first set bit in a 16 or 32-bit operand can be quickly determined from the left and right, respectively.

Table 3-9. *Bit Manipulation Instructions*

Mnemonic	Description	Mnemonic	Description
BSF	Bit scan forward	BTC	Bit test and complement
BSR	Bit scan reverse	BTR	Bit test and reset
BT	Bit test	BTS	Bit test and set

String Instructions

String instructions (Table 3-10) provide an efficient means of processing large operands that occur sequentially in memory, each of which may be one, two, or four bytes in size. By using the repeat instruction prefix, REP, a value loaded into ECX determines how many operands there are. Each of the string instructions performs its function on each operand sequentially in either the upward or downward direction, depending on the setting of the direction flag, DF.

Instruction REP MOVS, for example, fetches an operand from memory addressed by DS:ESI and stores it in another memory location addressed by ES:EDI. It then increments both ESI and EDI by the operand length, decrements ECX, and if the value of ECX is not zero, repeats the operation. If the direction flag is set, the string is decremented and the quantity by which ESI and EDI are altered on each instruction cycle is negative.

String instructions can also be used without the repeat prefix. In this case, only one instruction cycle is performed, and the ECX register is not modified.

Table 3-10. *String Instructions*

Mnemonic	Description (Use With REP Prefixes)
CMPSx	Compare string element
LODSx	Load byte/word/dword string element into AL/AX/EAX
MOVSx	Move string element from source to destination
SCASx	Scan string element for match against AL/AX/EAX
STOSx	Store AL/AX/EAX into string element

Control Transfer Instructions

There are many types of control transfer instructions, including jumps, calls, interrupts, and exceptions. Table 3-11 lists these instructions. The functions of some of them depend on the setting of the flags, which determine if they actually perform control transfers or behave as no-operation instructions, such as conditional instructions.

Jump instructions can be either conditional or unconditional. If conditional, the instruction includes a signed displacement, which is added to the address of the following instruction to determine the target instruction address. Unconditional jumps can also locate their target in this way, but they can also operate by selecting a register quantity instead of specifying a displacement.

Call and return instructions are similar to unconditional jumps, but they have the additional function of using the program stack to keep track of the return address.

Other forms of control transfer include intersegment jumps and calls, which allow execution to continue at a specific offset within a specific segment. These operations behave differently, depending on the mode of operation, and one should fully understand the segmentation rules of real and protected modes before attempting to use them.

Finally, some instructions perform control transfers by accessing the interrupt descriptor table. The INT 3 instruction is a good example. It fetches a new code segment and instruction pointer from the fourth entry in the IDT; stores its old code segment, instruction pointer, and flags on the stack; and begins execution in the new segment. Other instructions perform this function conditionally. INTO transfers control only if the overflow flag is set, and IDIV does so only if a divide exception is encountered.

Table 3-11. Control Transfer Instructions

Mnemonic	Description
CALL	Subroutine call or nested task switch
INT	Call to interrupt procedure
INTO	Call interrupt procedure on overflow
IRET	Interrupt return
Jcc	Conditional jump (e.g., JZ jumps if ZF is set)
JMP	Unconditional jump or non-nested task switch
LOOP	Loop [E]CX times
LOOPNZ	Loop [E]CX times or till ZF clear
LOOPZ	Loop [E]CX times or till ZF set
RET	Return from subroutine call
SETcond	Set byte on condition (e.g., SETZ set byte to 1 if ZF set)

Flag Instructions

Flag control instructions (Table 3-12) operate on the flags register, either the whole register or specific flags. The STC instruction sets the carry flag without altering any other bits. The POPF instruction reads an operand from the stack and stores it into the flags.

Table 3-12. Flag Instructions

Mnemonic	Description	Mnemonic	Description
CLC	Clear carry flag CF	POPF[D]	Pop into FLAGS/EFLAGS
STC	Set carry flag CF	PUSHF[D]	Push FLAGS/EFLAGS onto stack
CMC	Complement carry flag CF	SAHF	Store FLAGS into AH
CLD	Clear direction flag DF	STD	Set direction flag DF
CLI	Clear interrupt flag IF	STI	Set interrupt flag IF
LAHF	Load AH into FLAGS		

Segment Manipulation Instructions

The function of instructions that operate on segment registers varies, depending on the execution mode of the processor. A POP seg instruction, for example, will function differently in protected mode than in real mode.

Among the instructions in this set (Table 3-13) are PUSH and POP seg, MOV seg, and the load seg instructions (LxS). The LxS series of instructions is provided to load both a segment selector and a general register pointer simultaneously.

Table 3-13. *Load and Store Segment Instructions*

Mnemonic	Description	Mnemonic	Description
LDS	Load pointer to DS	LSS	Load pointer to SS
LES	Load pointer to ES	MOV sreg	Move to or from segment register
LFS	Load pointer to FS	POP sreg	Pop from stack to segment register
LGS	Load pointer to GS	PUSH sreg	Push onto stack

The rules listed in Table 3-14 should be observed when selecting a segment.

Table 3-14. *Segment Selection Rules*

Operation	Default Segment	Able to Override?
Code fetches	CS	No
Destination of string instructions	ES	No
Stack instructions	SS	No
Address generated with base from ESP or EBP	SS	Yes
All other address	DS	Yes

Protection Control Instructions

This group of instructions, listed in Table 3-15, establishes and maintains system protection features. LIDT loads the register that points to the interrupt descriptor table, which normally only occurs when entering the protected mode. LLDT loads the register that points to the local descriptor table.

Table 3-15. *Protection Control Instructions*

Mnemonic	Description
ARPL	Adjust requestor privilege level
LAR	Load access rights
LGDT	Load global descriptor table register
LIDT	Load interrupt descriptor table register
LLDT	Load local descriptor table register
LMSW	Load machine status word (see also MOV)
LSL	Load segment limit
LTR	Load task register and its shadow descriptor register
SGDT	Store global descriptor table register
SIDT	Store interrupt descriptor table register
SLDT	Store local descriptor table register
SMSW	Store machine status word (see also MOV)
STR	Store task register
VERR	Verify a segment for read access
VERW	Verify a segment for write access

Miscellaneous Instructions

This group consists of the instructions shown in Table 3-16. It includes two important instructions, NOP and LEA. The NOP instruction performs no function. The LEA instruction calculates an address, but rather than access the operand at that location, simply stores the calculated address in a general register.

Table 3-16. *Miscellaneous Instructions*

Mnemonic	Description
BOUND	Verify that value is in specified range
CLTS	Clear task-switched flag TS
ENTER	Enter nested procedure
HLT	Cease execution until interrupt detected
INT	Generate software interrupt
INTO	Generate INT 4 software interrupt if OF set
IRET[D]	Return from interrupt handler or nested task
LEA	Load effective address into general register
LEAVE	Leave nested procedure
NOP	No operation
WAIT	Wait for BUSY to deactivate
XLATB	Look up AL in translate table

Programming Guidelines

This section discusses some special uses of the general registers and suggests ways to optimize your code for the Super386 processor. See Appendix C for more advanced programming issues.

Register Usage

The eight general registers have different requirements because of their implicit use in different instructions. This uniqueness places important restrictions on their use. When the functions for which a register is dedicated are not needed in a particular instruction or sequence of instructions, the register can be used for other purposes.

EAX—This register has an instruction encoding that is one byte shorter for most operations, including ADD, XOR, and MOV. EAX must also be used as an operand for many instructions, including decimal arithmetic, multiply, and divide, as well as IN and OUT.

EBX—This is the most convenient register for generating addresses in 16-bit code.

ECX—This register is used both as a bit index in shift instructions and as an iteration count by LOOP and repeated string operations.

EDX—This register participates in multiply and divide operations, and specifies the port number for IN and OUT instructions.

ESI—The ESI register determines the source operand memory address for string instructions; can be used to index memory by 16-bit addresses.

EDI—This register determines the destination operand memory address for string instructions; can be used to index memory by 16-bit addresses.

EBP—This register points to the base of the stack.

ESP—The ESP register points to the top of the stack.

Optimizing Execution Speed

A dramatic improvement in execution speed can be achieved by following a few simple programming rules. Many instructions have been optimized to execute quickly on the Super386 processor. On the 38605 processor, many instructions are designed to take special advantage of the architecture of the instruction cache. The following are a few guidelines that will optimize your execution time:

- Favor Not-Taken Jumps—When conditional jumps are used, favor the not taken case. All not-taken jumps execute in one clock.
- Align Jump Targets—Align the target of jump instructions to doubleword boundaries. This increases the probability that the full target instruction will be available from the first instruction fetch.
- Align Operands—Align operands so that they do not cross doubleword boundaries. Unaligned operands require multiple bus accesses.
- Use One-Byte Displacement Jumps—Whenever possible, use one-byte displacement jump instructions. These are fast on all Super386 processors; the 38605 processor executes them more than twice as fast as their word or dword displacement counterparts.
- Interleave Memory Operations—Follow fetches or stores to slow memory, such as video displays, by unrelated instructions without memory operands. The processor can continue execution when no more than one memory access is pending.
- Shift Instructions—Use shift instructions when multiplying or dividing by powers of 2.
- Consider Timing of Register Loads—Avoid loading a register immediately before using it to generate an address. The processor stalls when this happens. Fetching the value two or more instructions before using the register will eliminate this delay.
- Avoid Loop Instructions—While loop instructions are convenient, the two-instruction sequence DEC/JNZ is significantly faster.
- Write 512-Byte Critical Routines for Instruction Cache—Write assembly language routines of no more than 512 sequential bytes, starting at an address that is a multiple of 16. If a program sequence fits entirely in the cache, sequential and near-jump instruction fetches will not interfere with operand accessing. As a rule, a routine of up to roughly 150 assembly language instructions can fit in the cache in protected mode (up to 200 instructions in real mode).

CHAPTER 4

System Programming

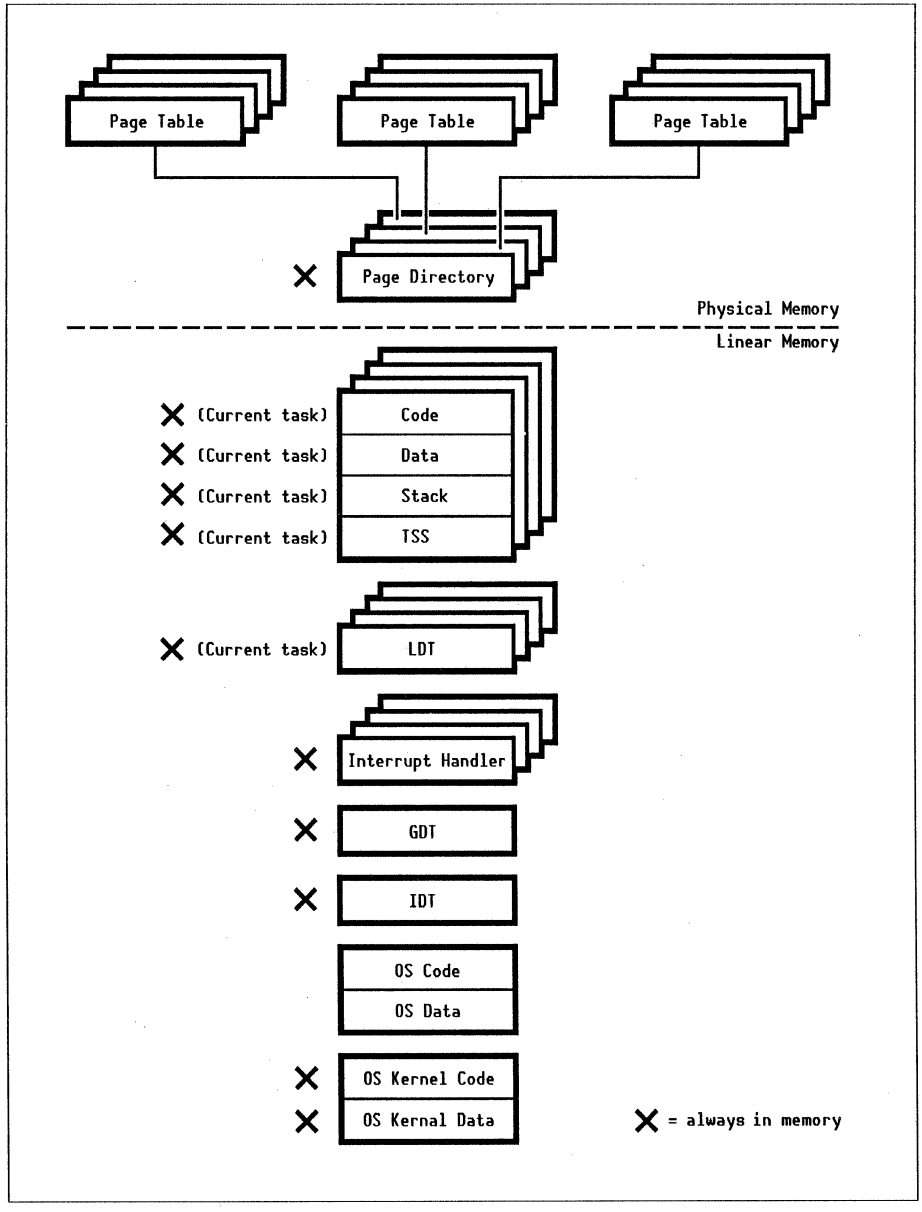
Figure 4-1 presents a broad overview of the types of data structures that an operating system can create in a full-featured system running in protected mode on the Super386 processor. Starting from the bottom of the figure and moving up, the data structures include the following:

- Operating System Kernel— Consists of code and data segments. (In this figure, the stack is in the data segment, but it could have a separate segment.)
- Operating System—Consists of code and data segments, similar to the kernel.
- Interrupt Descriptor Table (IDT)—Contains the control-gate descriptors for interrupts and exceptions.
- Global Descriptor Table (GDT)—Contains the control-gate descriptors and segment descriptors for code, data, and task segments that are available globally.
- Interrupt Handlers—Service interrupts and exceptions.
- Local Descriptor Tables (LDTs)—Contain the control-gate descriptors and segment descriptors for code and data segments that are available only to a specific task or set of tasks.
- Single Task or Program—Contains the following elements:
  - Code segment
  - Data segment (stacks are sometimes included with the data)
  - Stack segment
  - Task state segment (TSS), which stores a task's context during a task switch
- Page Directory—Contains entries that locate page tables.
- Page Tables—Contain entries that locate 4kB physical pages.

The SuperState V extensions include other resources that are not shown in this figure. They are used for power management and device virtualization. The SuperState V resources are transparent to existing operating systems, as described in the section entitled "SuperState V mode."

This chapter describes the methods of creating and maintaining the system data structures. It expands on the discussions in Chapter 2 by explaining all mechanisms from the viewpoint of system programming. It also includes sections on multitasking, protection mechanisms, testing, and debugging.

Figure 4-1. *System Data Structures*



System Registers

Before exploring the details of segmentation, paging, and multitasking, this section provides some background on the system registers. These registers are referred to frequently throughout the sections that follow.

Figure 4-2 shows the processor's register set. Some of these—the base register set and the segment registers—are visible to application software. Others are either visible only to system software or are invisible registers, called *shadow registers.* The system-level registers include the following:

- Flags register: EFLAGS
- Segment registers and shadow registers: CS, DS, SS, ES, FS, and GS
- System segment registers and shadow registers: TR and LDTR
- System address registers: GDTR and IDTR
- Control registers: CR3, CR2, and CR0
- Debug registers: DR7:6 and DR3:0
- Test registers: TR6 and TR7.

These registers are discussed in the following paragraphs.

Flags Register (EFLAGS)—In addition to the bits that can be changed by application programs, the EFLAGS register contains other bits that only the system software can change.

Segment and Shadow Registers (CD, DS, SS, ES, FS, GS)—The six 16-bit segment selector registers have invisible 64-bit shadow registers that are loaded automatically with the corresponding segment descriptors when the segment selectors are loaded.

System Segment and Shadow Registers (TR and LDTR)—The two 16-bit system segment registers contain system selectors: the task register (TR) references the current task state segment (TSS), and the local descriptor table register (LDTR) references the current local descriptor table (LDT). Both registers have invisible 64-bit shadow registers that are loaded automatically with the TSS and LDT descriptors when the TR and LDTR are loaded.

System Address Registers (GDTR and IDTR)—The 48-bit global descriptor table register (GDTR) and the interrupt descriptor table register (IDTR) reference the global descriptor table (GDT) and the interrupt descriptor table (IDT).

Control Registers (CR3, CR2, and CR0)—Control registers are 32-bit registers that are used to control and observe the status of segmentation, paging, task switching, and coprocessor operations.

Debug Registers (DR7:6 and DR3:0)—The 32-bit debug registers are used to debug programs.

Test Registers (TR6 and TR7)—Two 32-bit test registers, TR6 and TR7, are used to test the translation lookaside buffer (TLB).

Figure 4-3 illustrates how some of these registers relate to data structures. In this figure, the arrows show how the content of a register is used to access a data structure, or how entries in a data structure (such as descriptors in a descriptor table) are used to access other data structures. For example, the CS selector register, when loaded by software with a code segment selector, points to a code segment descriptor in either the LDT or GDT. This descriptor, in turn, locates the associated code segment.

These registers and relationships are described in this section and in later sections entitled "Segmentation", "Multitasking,", "Testing the TLB", and "Debug Control and Status,"

Figure 4-2. System Registers

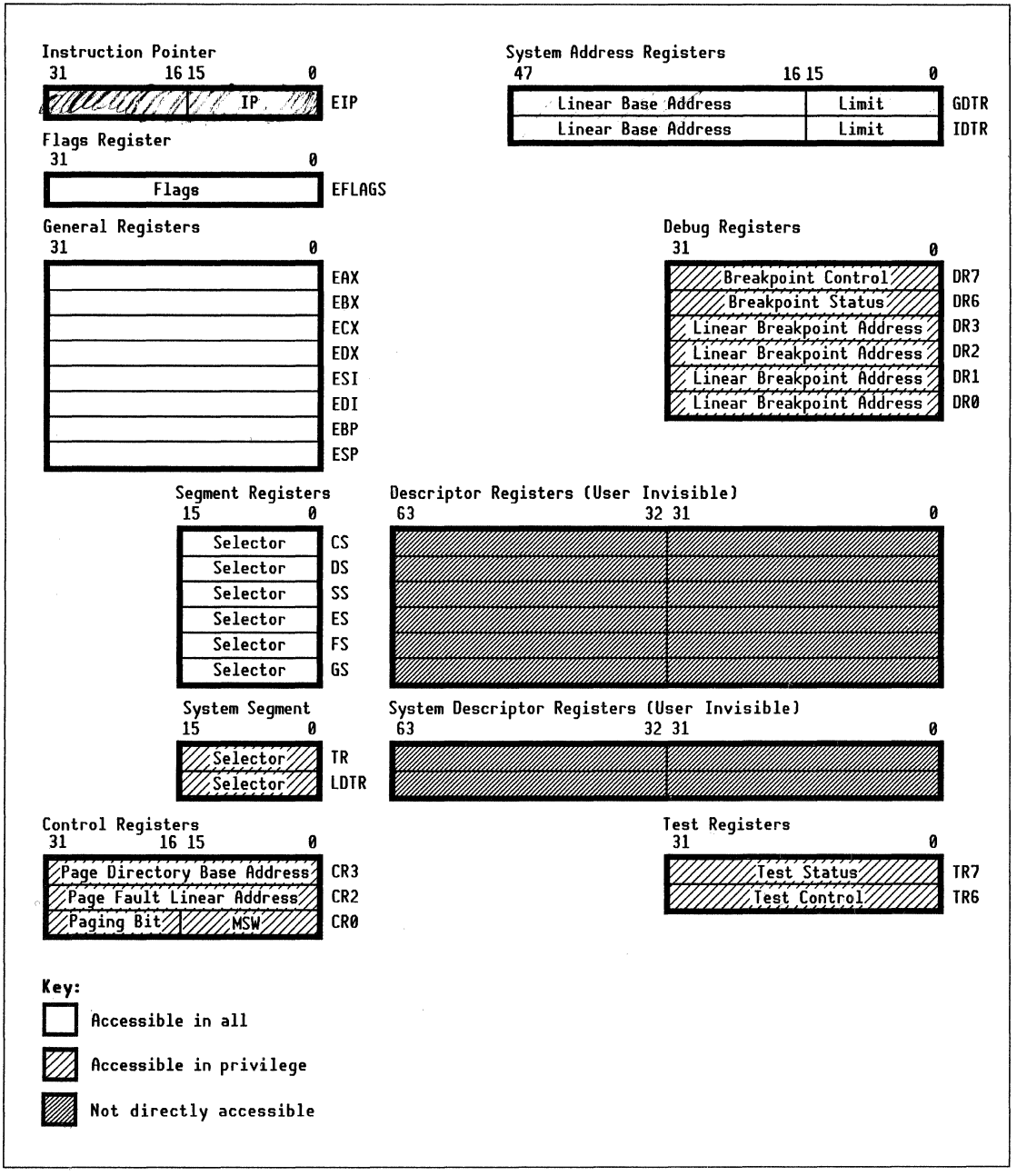
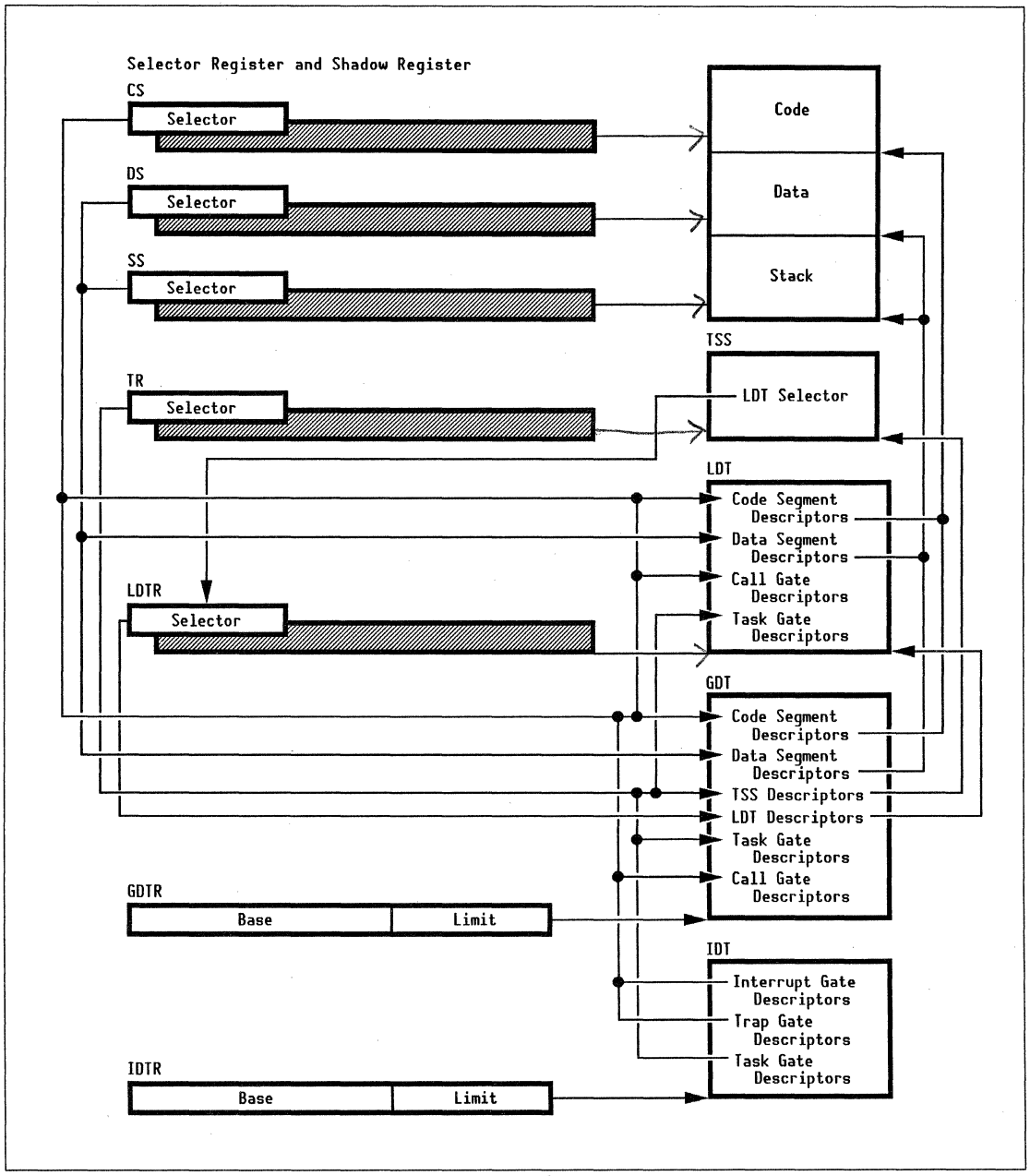


Figure 4-3. Registers Associated With Segments and Tables



In addition to these relations, all selectors specify either the GDT or LDT (TI bit)

Compare Table 4-3 p. 4-23
" 4-4 p. 4-40

General Registers

Figure 4-4 shows the general registers. When a general register is pushed on or popped from the stack, the default operand size is specified by the D bit in the code segment descriptor (see the section "Segment Descriptors"). If a destination register has more bytes than the operand, the upper part of the register is left unchanged.

The binary sort order for instruction decoding is show in Figure 4-4.

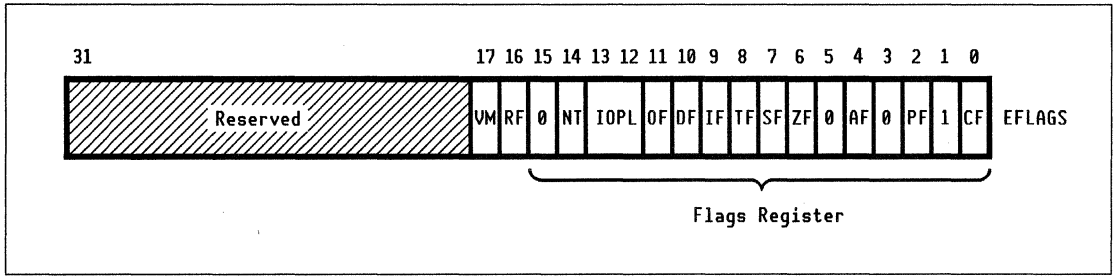
Figure 4-4. *General Registers*

Register Type	8-bit Registers	8-bit Registers	16-bit Registers	32-bit Registers	Binary Sort Order
	31 16	15 8	7 0		
General Purpose	AH	AL	AX	EAX	000
General Purpose	BH	BL	BX	EBX	011
General Purpose	CH	CL	CX	ECX	001
General Purpose	DH	DL	DX	EDX	010
Source Index			SI	ESI	110
Destination Index			DI	EDI	111
Stack-Frame Base Pointer			BP	EBP	101
Stack Pointer			SP	ESP	100

Flags Register (EFLAGS)

The 32-bit flags register, shown in Figure 4-5, has only the lower 18 bits defined. The lower 16 bits constitute the 8086 flags register. Most of these bits reflect status after an operation. Bits 17 and 16 enable virtual-8086 mode and control repeated breakpoints. Chapter 2 describes the flags available to application software. The description following Figure 4-5 describes all flags represented in the flags register in greater detail.

Figure 4-5. EFLAGS Register



bits: 17 VM — Virtual-8086 Mode—This bit indicates whether the processor is in virtual-8086 mode. Protected mode must be enabled (PE bit set to 1 in CR0) for this bit to have an effect, because virtual-8086 mode is a sub-mode of protected mode:

1 Enable virtual-8086 mode
0 Disable virtual-8086 mode.

A general-protection fault is generated when executing a privileged opcode in this mode. The VM bit can be set only in protected mode, either with the IRET instruction from privilege level 0 or by a task switch.

Note that the VM bit bears no relation to virtual memory, as the acronym might imply. The VM relates only to multitasking of 8086 programs.

16 RF — Resume Flag—This bit indicates whether breakpoint debugging should be resumed after a breakpoint is encountered. When set to 1, it ensures that restarted instructions do not generate repeated debug faults.

Instead, a debug fault is ignored for one instruction:

1 Ignore breakpoint for one instruction.
0 Do not ignore breakpoint.

Because RF is in the EFLAGS register, it is loaded whenever an IRET instruction is performed. When the interrupt or exception handler returns, it must do so with the IRETD instruction to pop all 32 flag bits, including the RF. The RF is not affected by the POPFD and IRET instructions. It is set according to the EFLAGS memory image after an IRETD instruction is performed, and after JMP, CALL, and INT instructions have caused a task switch.

15	—	Reserved—This bit is cleared to 0.
14	NT	Nested Task—This bit indicates whether the current task is nested within another task. It only applies to protected mode. If an IRET instruction is executed with NT set to 1, the current task state is saved and a task switch is performed to the task that invoked the current task. The back-link field in the current task state segment (TSS) is used to access the old task. If the IRET instruction executes successfully, NT is cleared to 0. A CALL or INT instruction that causes a task switch sets it to 1. 1 Task is nested 0 Task is not nested.
13:12	IOPL	I/O Privilege Level—These two bits determine the I/O privilege level required to perform I/O instructions. They apply only to protected mode: 11 Privilege level 3 (lowest) 10 Privilege level 2 01 Privilege level 1 00 Privilege level 0 (highest). In protected mode, if the current privilege level (CPL) is numerically greater than the IOPL, the I/O permission bitmap (IOPB) is interrogated. In virtual-8086 mode, the IOPB is interrogated for any IOPL. The IOPL also determines the maximum CPL value allowed to alter the interrupt enable (IF) flag by following a pop into the EFLAGS register. POPF and IRET instructions can alter the IOPL bits when they are executed from privilege level 0. A task switch always alters the IOPL bits when the new image of the flags is loaded from the new task state segment (TSS).
11	OF	Overflow Flag—This bit indicates whether the uppermost bit (sign bit) of an operand is changed as a result of an operation. 1 Overflow 0 No overflow.

10	DF	Direction Flag—Bit DF indicates whether a source or destination address pointer of a string instruction (the contents of the ESI and/or EDI register) should be incremented or decremented after each iteration of the instruction execution. The increment is +1, +2, or +4 and the decrement is -1, -2, or -4, depending on the operand size. The flag can be explicitly set or cleared by the STD and CLD instructions. 1 Decrement 0 Increment.
9	IF	Interrupt Enable Flag—This bit indicates whether external interrupt requests (INTR signal) are to be recognized. The flag can be explicitly set or cleared by the STI and CLI instructions. IOPL indicates the maximum CPL value allowed to alter the IF flag: 1 Enable INTR. 0 Disable INTR.
8	TF	Trap Flag—This bit indicates whether a single step debug trap (exception 1) should be generated. 1 Trap on single steps 0 Do not trap on single steps.
7	SF	Sign Flag—Bit SF indicates whether an arithmetic operation had a positive or negative result, as indicated by the high-order bit of a byte, word, or doubleword (bit 7, 15, or 31): 1 Negative result 0 Positive result.
6	ZF	Zero Flag—This bit indicates whether an arithmetic operation resulted in zero: 1 Zero result 0 Nonzero result.
5	—	Reserved—This bit is cleared to 0.
4	AF	Auxiliary Flag—This bit indicates whether an arithmetic operation resulted in a carry out (addition) or borrow (subtraction) from bit 3 of the least-significant byte, regardless of the operand size. It is useful for BCD arithmetic. 1 A BCD carry out or borrow occurred. 0 No BCD carry out or borrow occurred.

3	—	Reserved—This bit is cleared to 0.
2	PF	Parity Flag—PF indicates the number of 1s in the low-order operand byte after an arithmetic operation, regardless of the operand size: 1 Even number of 1s 0 Odd number of 1s.
1	—	Reserved—This bit is set to 1.
0	CF	Carry Flag—This indicates whether an arithmetic operation resulted in a carry (addition) or borrow (subtraction) beyond the high-order bit of the operand. The flag can be set explicitly or cleared by the STC and CLC instructions. The flag can be complemented with the CMC instruction: 1 A carry out or borrow occurred. 0 No carry out or borrow occurred.

use definition on p. 2-32

Control Registers (MSW and Paging Control)

The following four 32-bit control registers, shown in Figure 4-6, contain the paging controls and the machine status word (MSW):

Associated w/ a table (stored in TSS)

- CR3—Page directory base address
- CR2—Page fault linear address
- CR1—Reserved
- CR0—Page enable and machine status word (MSW).

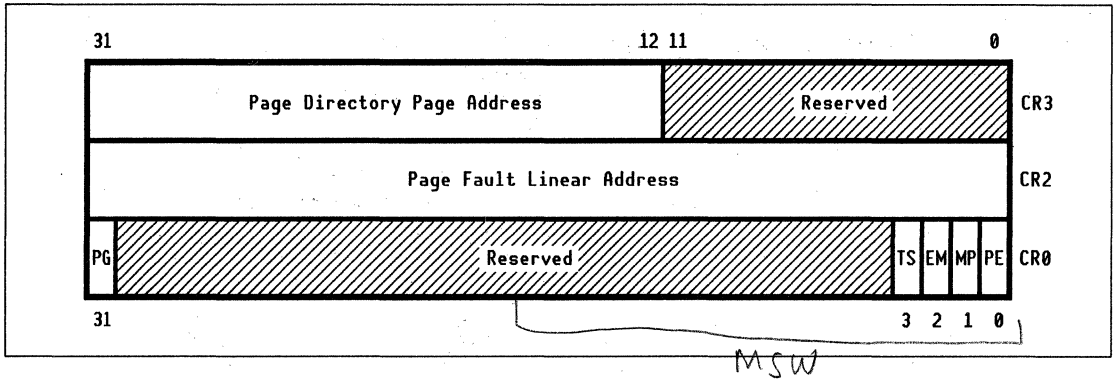
These registers are discussed in the following paragraphs.

CR3, Page Directory Base Address—When paging is enabled in protected mode, CR3 holds the most significant 20 bits of the page directory. The 12 lower significant bits are ignored. CR3 is changed automatically during a task switch if the new task has a different page directory.

CR2, Page Fault Linear Address—If a page-fault exception occurs, the processor stores the 32-bit linear address that caused the exception in CR2. This address can be used by the page-fault exception handler to determine which page to load from mass storage.

CR0, Page Enable and Machine Status Word—Paging enable is the high-order bit. The lower 16 bits are the MSW, which is used for mode control, task switching, and coprocessor monitoring. The bits are defined following Figure 4-6.

Figure 4-6. Control Registers



31	PG	Page Enable—This bit enables paging, which allows the virtual memory space of 4GB to be logically allocated among 4kB pages in physical memory. It can only be set in protected mode (PE = 1), and it must be set in virtual-8086 mode if more than one program will be run in that mode. A jump instruction must be executed to clear the instruction pipeline after changing this bit. 1 Paging enabled 0 Paging disabled.
3	TS	Task Switched—Bit TS is set automatically whenever a task switch is performed. It can be tested by a task to determine whether a previous task may have had control of the coprocessor. The bit can be cleared with the CLTS instruction. When TS is set to 1, a coprocessor instruction (ESC opcode) will cause a Coprocessor Not Available trap (exception 7). If both the TS and MP bits are set, a WAIT instruction will also generate this exception. 1 Task switch occurred since last cleared. 0 Task switch has not occurred since last cleared.
2	EM	Emulate Coprocessor—When this bit is set, all coprocessor instructions generate a Coprocessor Not Available trap (exception 7). The exception handler can then emulate the coprocessor instruction. 1 Exception handler emulates math opcode. 0 No emulation.

1	MP	Math Coprocessor Present—The MP bit is used in conjunction with the task switched (TS) bit to synchronize the processor with a math coprocessor. It determines whether a WAIT instruction will generate a Coprocessor Not Available trap (exception 7). 1 Coprocessor present 0 Coprocessor not present.
0	PE	Protection Enable—This bit selects protected mode or real mode. If paging is enabled (PG = 1), protection must also be enabled; otherwise, exception 13 is generated. A jump instruction must be executed to clear the instruction pipeline after changing this bit. 1 Protected mode 0 Real mode.

The processor is initialized in the real mode with both the PG and PE bits of CR0 cleared to 0. The CR3 and CR0 registers can be loaded with MOVE instructions (such as MOV CR0, reg), although the LMSW and SMSW instructions can also be used to load CR0. After the PE bit is changed to its desired value, a JMP instruction will clear the pipeline of any instructions that have been fetched.

Segmentation

Chapter 2 provides an overview of how segmentation partitions logical addresses into linear address segments up to 4GB (2^{32} bytes) in size. Segment registers hold segment selectors, which reference segments via segment descriptors located in a descriptor table. An instruction making a memory access references a segment selector, and thereby indirectly locates the memory segment.

Several segmentation models can be implemented. Flat models map all segments to the same linear memory, thereby effectively disengaging the segmentation mechanism. UNIX® and other paged but non-segmented operating systems use this environment. Multisegment protected models map segments into discrete, limited parts of the memory, thereby isolating one segment from another and avoiding areas of the linear address space that are not populated with RAM or ROM hardware.

It is possible and sometimes desirable to have two or more segments share the same location in memory (that is, to have segments overlapping). For example, ROM addresses often hold both code and data. These designs, as well as complex segmented and demand-paged designs, can be supported within the framework of the processor's segmentation and paging architecture.

Segment Registers and Their Shadows

The segment registers contain segment selectors. Some of the registers are reserved for segment selectors of a specific type, as shown in Table 4-1. Each 16-bit segment register has a corresponding 64-bit shadow register, invisible to software, which holds the segment descriptor corresponding to its selector. When a selector is loaded into a segment register, the segment descriptor is also loaded automatically into the segment register's shadow register.

Table 4-1. *Types of Segments*

Segment	Functon	Required? (Protected Mode)	Required? (Real Mode)
CS	Code	Yes	Yes
DS	Data and/or stack	Yes, to read or write data.	Yes, to read or write data.
SS	Stack	Yes, if DS is used for stack, copy the DS selector to SS.	Yes, to perform stack operations and handle interrupts. If DS is used for stack, copy the DS selector to SS.
ES	Extra	Required for MOVS, CMPS, and STOS instructions. Initialize to zero if not used.	Required for MOVS, CMPS, and STOS instructions. Initialize to zero if not used.
FS	Extra	No, initialize to zero if not used.	No, initialize to zero if not used.
GS	Extra	No, initialize to zero if not used.	No, initialize to zero if not used.

It is possible to support aliases using segmentation. Code segments, which normally store only executable code, may also store data if a data segment is mapped to the same address space as the code segment. This is useful for ROM, which may need to hold constants as well as code. It is also possible to write to code segments with the same data-mapping arrangement. Protected mode prohibits modifications of the code segment. However, by aliasing the code and data segments, a write to the data segment will update the identical location in the code segment. It is possible to implement partially overlapping aliases as well. The stack segment, for example, may begin in the middle of the data segment and extend to the end of the data segment. This makes the stack segment a subset of the data segment, while still allowing the data segment direct access to the stack.

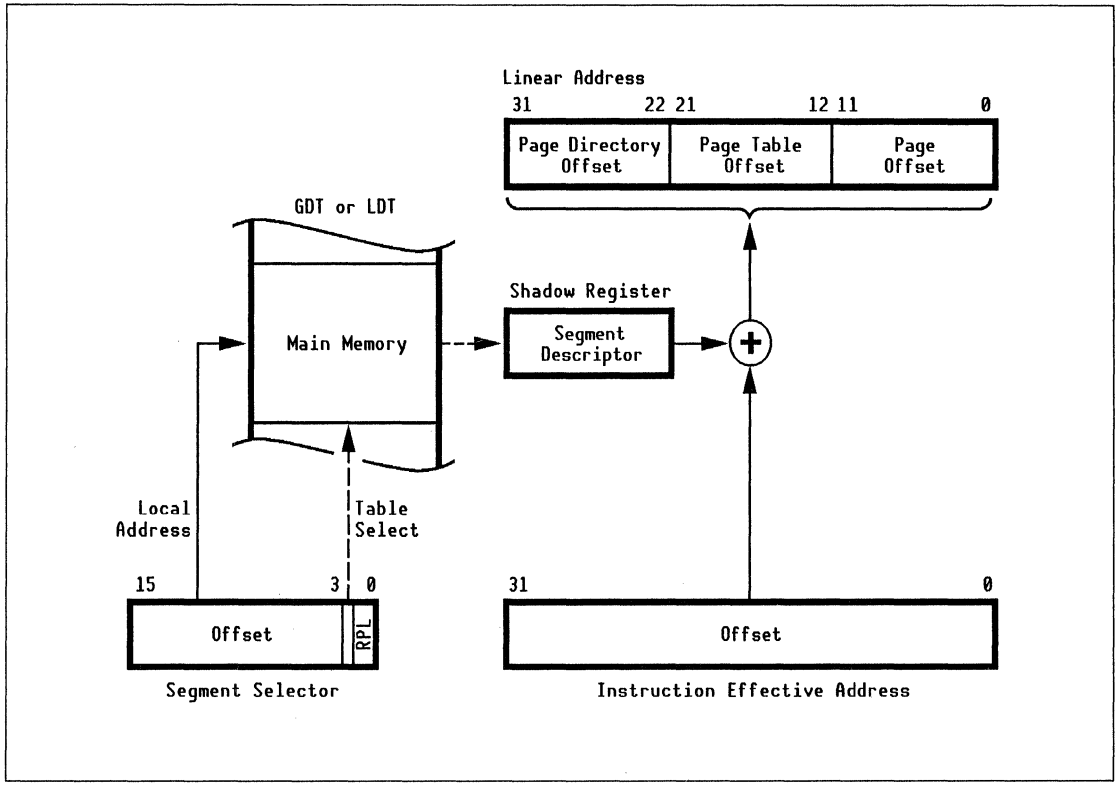
A special type of code segment, called a *conforming code segment*, is defined for libraries, interrupt and exception handlers, and other types of code shared by many applications. For conforming code segments, privilege-level checks are not enforced; any privilege level can call or jump to such a segment. See the section entitled "Conforming Code Segments" later in this chapter.

addr are scaled by 2^3 bits since descriptors are 8 bytes long.

Segment Selectors

In protected mode, the 16-bit segment selector contains a 13-bit offset that points to a segment descriptor in the GDT or LDT. The segment descriptor defines the base address, limit, and other properties of the segment. Figure 4-7 shows the mechanism.

Figure 4-7. *Segmentation Mechanism*

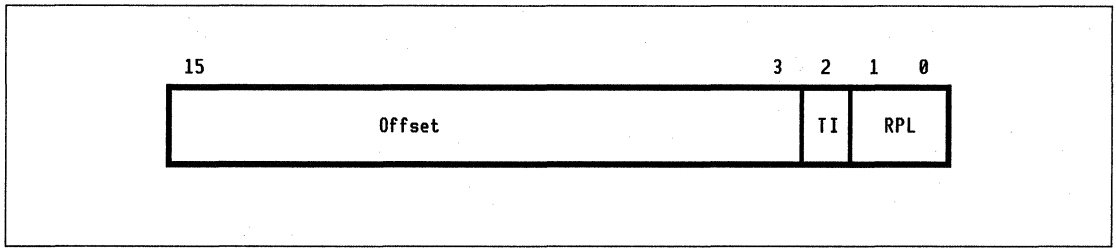


Logically, they must be created in appl. software

Segment selectors in protected mode have three functions: they indicate which table contains the segment descriptor of interest, they index the descriptor in that table, and they establish a requestor privilege level (RPL) for any activity relating to that selector. Selectors are normally assigned by the operating system. Application programs may see them in pointer variables. Figure 4-8 shows the format in protected mode.

The selectors of destination segments are examined by the processor in its privilege checks. How is this done? Where does the processor look?

Figure 4-8. Segment Selector



bits:	15:3	Offset	Descriptor Table Offset—These bits are indexes into the descriptor table defined by the TI bit. The base of the table is contained in the GDTR or LDTR. A *null selector* is defined as one which has all zeros in this field.
	2	TI	Table Indicator—This bit indicates whether the LDT or GDT contains the descriptor. 1 Local descriptor table (LDT) 0 Global descriptor table (GDT).
	1:0	RPL	Requestor Privilege Level—The RPL bit represents a privilege level used to override the CPL when a descriptor is loaded. The RPL is normally used by the operating system to "weaken" (raise the privilege level number of) the effective CPL at which a code segment executes. 11 Privilege level 3 (lowest) 10 Privilege level 2 01 Privilege level 1 00 Privilege level 0 (highest). The RPL can be updated with the ARPL instruction. When loading code segments, the RPL value in the CS register is automatically overwritten by the processor with the CPL after a privilege-level check is performed on the load operation and the code segment is loaded.

The RPL weakens the CPL during the loading of a descriptor, when the descriptor's DPL is checked for valid access. The section entitled "Protection Mechanisms" explains the CPL, DPL, and RPL checking rules. The CPL is loaded only into the RPL field of the CS selector after the privilege checks for access are performed and the selector is loaded. The operating system can examine the CPL by storing the CS selector into a general register or memory.

In real mode, the segment register contains a 16-bit selector that is shifted to the left by four bits to form a 20-bit base address for the segment. The result is then stored as the 32-bit segment base address in the corresponding shadow register, whose upper 12 bits are filled with zeros. The limit field is left unmodified, as are the other properties of the segment. The limit field is set to 64kB on reset. Therefore, loading a selector in real mode also loads the 32-bit segment base address in the shadow register but leaves the remaining 32 bits of the shadow register unmodified. The RPL is not defined.

Segment selectors are loaded with a move, pop, load full pointer, far jump, far call, interrupt or exception, or a return from an interrupt or procedure. A return to an originating segment requires a reload of its selector. In using these instructions, the default segment register for data references is dependent on the base register used. The DS register is the default segment register for all selected base registers (including no base register) except for the ESP or EBP base register. If the ESP or EBP base register is selected, the default register is SS. ESP cannot be used as an index register. The choice of default segment register is not affected by using EBP as an index register.

The default segment register can be overridden by using segment prefixes. However, the implied segment selection used by string destinations and PUSH and POP instructions cannot be overridden. In these cases, segment prefixes are ignored.

Segment Descriptors

Segment descriptors define the base address, limit, attributes, and access rights of a segment. These elements are discussed in the following paragraphs.

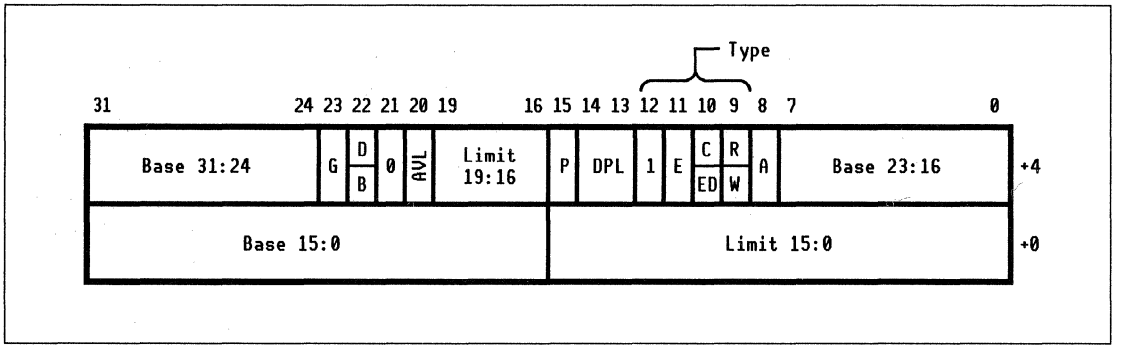
Base Address—The base address is the starting address of the segment in the linear address space.

Limit—The limit defines the upper bound for the byte effective address in this segment or a lower bound in an expand-down segment. The target address is defined by the base address plus an offset provided in the instruction. In expand-up segments, the offset must not exceed the limit. In expand-down segments, the offset must exceed the limit. During a reset, which initiates real mode, the limit is set to 64kB.

Attributes and Access Rights—Attributes and access rights are segment characteristics such as code or data; default address and operand size; expand-up or expand-down, accessed, conforming or non-conforming code; the privilege level required for access; the presence of the segment in memory; and its read/write access availability.

Segment descriptors are stored in memory in descriptor tables, which are arrays of segment descriptors. The segment selector identifies a segment by specifying the location of the descriptor within the descriptor table. Figure 4-9 shows the memory image of a descriptor. Stack segments are data (vs. code) segments.

Figure 4-9. *Segment Descriptors*



(+4 is high dword, +0 is low dword)

31:24 7:0 31:16	(+4) (+4) (+0)	Base	Segment Base Address—These bits contain the 32-bit linear address of the segments's base memory.
23	(+4)	G	Granularity—The G bit determines the maximum segment size (limit): 0 Byte-granular limit; the maximum segment size is 2^{20} bytes. 1 Page-granular limit; the maximum segment size is 2^{32} bytes. When the G-bit is set to 0, the 20-bit limit value, limit 19:0, is zero-extended to 32-bits. This provides the byte-granular limit. When the G-bit is set to 1, the 20-bit limit value is shifted left by 12 bits and OR'd with 0FFFh, thus providing a 32-bit limit value that is page-granular.

22	(+4)	D/B	Default Size or Upper Bound (Big bit)—For code segments (E bit = 1), the D bit indicates the default address and operand size. For the stack segments, the B bit (sometimes called the big bit) controls whether the stack address size is 16 bits or 32 bits. For expand-down stack segments or any other type of expand-down data segments (E bit = 0 and ED = 1), the B bit indicates the upper bound for the segment. The bit is ignored in all other cases. See Table 4-2 for the relationship between the D/B, E, C/ED, and R/W bits.
20	(+4)	AVL	Available to Software—This bit may be used by system software. It is not interpreted by the processor.
19:16 15:0	(+4) (+0)	Limit	Segment Limit—The segment limit is expanded to 32 bits by interpreting the granularity (G) bit.
15	(+4)	P	Present—If set, this attribute indicates that the descriptor is present in memory and is therefore valid. If this bit is clear, an attempt to access the segment causes an exception. 1 Present (valid) 0 Not present (invalid).
14:13	(+4)	DPL	Descriptor Privilege Level—Bit DPL indicates the privilege level of the descriptor. The processor uses the DPL to determine access rights to the segment pointed at by the descriptor. 11 Privilege level 3 (least privileged) 10 Privilege level 2 01 Privilege level 1 00 Privilege level 0 (most privileged). Note: Bits 12:8 of the upper dword are often referred to as the *type field.*
11	(+4)	E	Executable—This bit indicates whether the segment contains code (which cannot be written) or data (which can be written). See Table 4-2 for the relationship between the D/B, E, C/ED, and R/W bits.

10 (+4) C/ED — Conforming/Expand Down—For code segments (E bit = 1), the C bit indicates whether the segment is conforming or nonconforming. For data or stack segments (E bit = 0), the ED bit indicates whether the segment expands up or down. For expand-down segments (such as stacks) the B bit specifies the upper bound. Table 4-2 shows the relationship between the D/B, E, C/ED, and R/W bits.

9 (+4) R/W — Read/Write—For code segments (E bit = 1), the R bit indicates whether the segment is readable. For data or stack segments (E bit = 0), the W bit indicates whether the segment is writable. Table 4-2 shows for the relationship between the D/B, E, C/ED, and R/W bits.

8 (+4) A — Accessed—The processor sets this bit when the segment is loaded. System software can clear the bit to 0 before running a program to determine whether the segment was loaded. After loading, read/write access to the segment can be determined by examining the dirty and accessed bits in the page directory and page tables.

1 Segment was read or written.
0 Segment was not read or written.

Table 4-2. *Relation of the D/B, E, C/ED, and R/W Fields*

Bit Bit Number	D/B 22	E 11	C/ED 10	R/W 9	Type of Segment
Code Segments	0	1	0	0	Nonconforming, nonreadable, default size = 16 bits
	1	1	0	0	Nonconforming, nonreadable, default size = 32 bits
	0	1	0	1	Nonconforming, nonreadable, default size = 16 bits
	1	1	0	1	Nonconforming, nonreadable, default size = 32 bits
	0	1	1	0	Conforming, nonreadable, default size = 16 bits
	1	1	1	0	Conforming, nonreadable, default size = 32 bits
	0	1	1	1	Conforming, readable, default size = 16 bits
	1	1	1	1	Nonconforming, readable, default size = 32 bits
Data Segments	X[1]	0	0	0	Expand up, nonwritable
	X[1]	0	0	1	Expand up, writable
	0	0	1	0	Expand down, nonwritable, upper bound = FFFFh, lower bound = limit
	1	0	1	0	Expand down, nonwritable, upper bound = FFFFFFFFh, lower bound = limit
	0	0	1	1	Expand down, writable, upper bound = FFFFh, lower bound = limit
	1	0	1	1	Expand down, writable, upper bound = FFFFFFFFh, lower bound = limit
Stack Segments[2]	0	0	0	0	Expand up, nonwritable, 16-bit stack address.[3]
	1	0	0	0	Expand up, nonwritable, 32-bit stack address.[4]
	0	0	0	1	Expand up, writable, 16-bit stack address.[3]
	1	0	0	1	Expand up, writable, 32-bit stack address.[4]
	0	0	1	0	Expand down, nonwritable, 16-bit stack address.[3] Upper bound = FFFFh, lower bound = limit.
	1	0	1	0	Expand down, nonwritable, 32-bit stack address.[4] Upper bound = FFFFFFFFh, lower bound = limit.
	0	0	1	1	Expand down, writable, 16-bit stack address.[3] Upper bound = FFFFh, lower bound = limit.
	1	0	1	1	Expand down, writable, 32-bit stack address.[4] Upper bound = FFFFFFFFh, lower bound = limit.

1 X = Don't care.

2 The B bit (bit 22) determines the stack address size for all stack operations: 0 = 16-bit addresses; 1 = 32-bit addresses.

3 A 16-bit stack address implies that all implicit stack references will be 16-bit operations.

4 A 32-bit stack address implies that all implicit stack references will be 32-bit operations.

Descriptor Tables and Their Registers

In protected mode, segment descriptors define all memory areas available to programs. These descriptors are located in one of the following tables in memory:

- Global descriptor table
- Local descriptor table
- Interrupt descriptor table.

The tables are described in the following paragraphs.

Global Descriptor Table (GDT)—The GDT can hold all types of descriptors, except descriptors for interrupt gates and trap gates. Descriptors are selected from the table by the 13-bit offset in the segment selector. There can be only one GDT. It is required and must be kept in memory at all times.

only used by tasks. A task switch is the only way to load.

Local Descriptor Table (LDT)—The LDT holds descriptors for code segments, data segments, call gates, and task gates associated with a task. Descriptors are selected from the table by the 13-bit offset in the segment selector. Only the current one needs to be kept in memory. LDTs are optional. A task's LDT selector is stored in its task state segment (TSS).

Interrupt Descriptor Table (IDT)—The IDT holds descriptors for interrupt gates, trap gates, and task gates. Descriptors are selected from the table by the interrupt or exception vector. There is only one IDT. It is required and must be kept in memory at all times.

Table 4-3 distinguishes the three types of descriptor tables. Descriptor tables are set up and maintained by the operating system and referenced by the processor. The tables stored in memory should be accessible only by the operating system.

Compare Table 4-4 (p. 4-40) Fig 4-3 (p. 4-6)

Table 4-3. *Descriptor Table Characteristics*

Condition		GDT	LDT	IDT	What They Point To
Maximum Number Possible		1	2^{13}-1	1	—
Minimum Number Required		1	0	1	—
Maximum Size and Content		2^{13}-1 eight-byte entries (first entry is null).	2^{13} eight-byte entries.	2^{8} eight-byte vectors in protected mode. 2^{8} four-byte vectors in real mode.	—
Segment Descriptors	Code Segments	X	X		Code segment
	Data/Stack Segments	X	X		Data or stack segment
	Task State Segments	X			Task state segment (TSS)
LDT Descriptors		X			LDT
Gate Descriptors	Call Gates	X	X		CS descriptor
	Task Gates	X	X	X	TSS descriptor
	Interrupt Gates			X	CS descriptor
	Trap Gates			X	CS descriptor

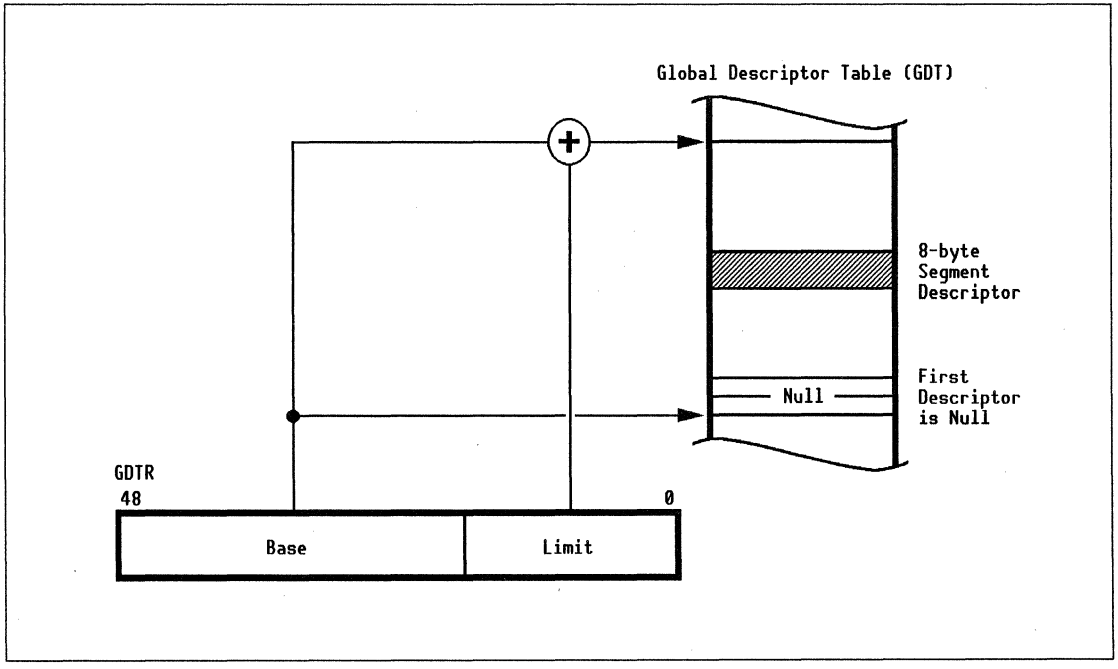
If paging is enabled, the descriptors required by the page-fault handler must be kept in memory. Other descriptors can be paged out of memory. For example, if the IDT points to the page fault handler through a descriptor in the GDT, those entries in the GDT and IDT must be present in memory. The operating system may do this by keeping the first 4kB page of each table in memory and locating the descriptors for the page fault handler in that page. If the IDT spans two pages, both must reside in memory.

Global Descriptor Table and Register

Segments shared by many procedures and tasks in the system are mapped by the global descriptor table (GDT), which is an array of segment and control-gate descriptors. The segments mapped by the GDT typically include those of the operating system. One system register is associated with the table: the 48-bit global descriptor table register (GDTR). The register contains the 32-bit linear base address and 16-bit limit of the GDT. Figure 4-10 shows the mechanism. The GDTR is a system address register. This type of register is not loaded with a segment descriptor, and the table is not defined as a segment.

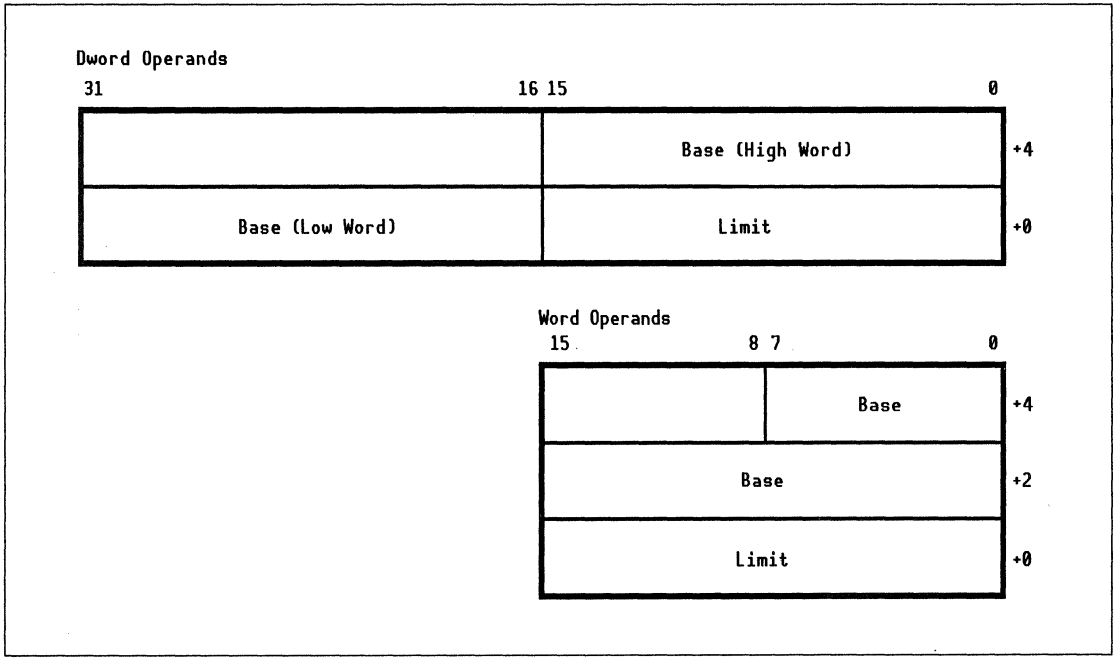
The operating system must load the lowest-order descriptor slot with a null (zero) descriptor. The table can contain up to 8k-1 descriptors, eight bytes each, plus the null descriptor in the first slot, for a total size of 64kB. The processor never accesses the null descriptor. A memory reference to the null descriptor will raise an exception. The gates contained in the GDT are described later in the section entitled "Control Gates and System Calls."

Figure 4-10. *GDT and GDTR*



The GDTR is loaded with the LGDT instruction. The argument passed in this load instruction is a memory data structure consisting (from low to high addresses) of a limit and base. Figure 4-11 shows the memory image of the argument, which has the same form for the GDTR and the IDTR. For 32-bit operands, a two-byte limit is followed by a four-byte base address. For 16-bit operands, a two-byte limit is followed by a three-byte base address, and the upper byte of the last word is not used. The SGDT instruction stores this value.

Figure 4-11. *GDTR and IDTR Memory Images*

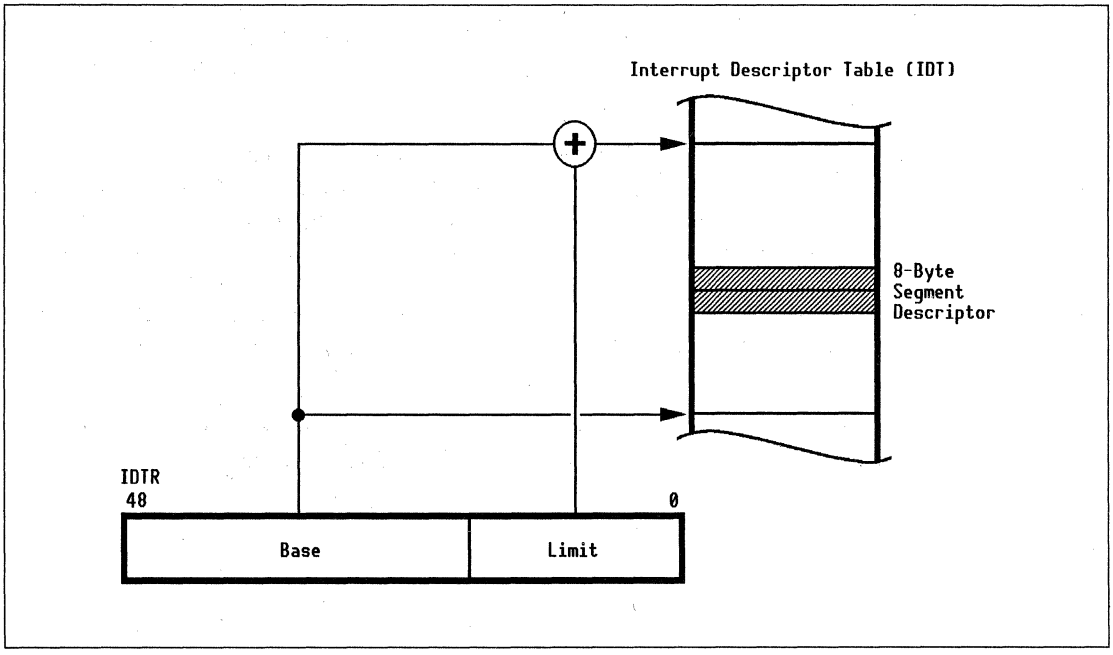


Interrupt Descriptor Table (IDT) and Register

The interrupt descriptor table (IDT) is an array of interrupt, trap, and task gate descriptors. The interrupt and trap gate descriptors hold a far pointer to an interrupt handler. The task gate descriptor facilitates a task switch to an interrupt handler.

One system register is associated with the table: the 48-bit interrupt descriptor table register (IDTR). The IDTR contains the 32-bit linear base address and 16-bit limit of the IDT. Figure 4-12 shows the mechanism. The IDTR, like the GDTR, is a *system address register*. This type of register is not loaded with a segment descriptor, and the table is not defined as a segment.

Figure 4-12. *IDT and IDTR*



The IDT has a structure similar to the global descriptor table, except that all descriptor slots of the IDT, including the first slot, may contain valid (non-null) descriptors. The table can have up to 256 entries, one for each vector. Each entry has the standard descriptor size of eight bytes. When indexing into the IDT, the vector is scaled by 8, the number of bytes in each descriptor.

The gates contained in the IDT are described in the section entitled "Control Gates and System Calls."

The IDTR is loaded with the LIDT instruction in real mode. The argument passed in these load instructions is a memory data structure consisting (from low to high addresses) of a limit and base. Figure 4-11 shows the memory image of the argument, which is the same form for both the GDTR and the IDTR. For 32-bit operands, a two-byte limit is followed by a four-byte base address. For 16-bit operands, a two-byte limit is followed by a three-byte base address, and the upper byte of the last word is not used. The SIDT instruction stores this value.

Local Descriptor Table (LDT), Register (LDTR), and Descriptor

The local descriptor table (LDT) contains descriptors used by a specific task, or by the programs that run under that task. These descriptors may include code and data segment descriptors, call gates, and task gates. The structure of an LDT is similar to that of the GDT, except that all descriptor slots of the LDT (including the first slot) may contain valid (non-null) descriptors. The table can contain up to 8k descriptors, eight bytes each, for a total size of 64kB.

The LDT is unlike the GDT and IDT in that the LDT is defined as a segment, with a segment descriptor, whereas the GDT and IDT are simply located by the base and limit contained in the GDTR and IDTR, respectively. The selector for the LDT segment descriptor is stored in the LDT field of the task's task state segment (TSS). During a task switch, this selector is loaded into the local descriptor table register (LDTR), which points to an LDT segment descriptor in the GDT. The GDT contains all LDT segment descriptors. Figure 4-13 shows the mechanism.

Several tasks can share a common LDT, so the same set of segments is available to all of these tasks. Two tasks can also have a descriptor for a shared segment in both of their LDTs; the descriptor does not have to be put in the GDT.

Figure 4-13. LDT and LDTR

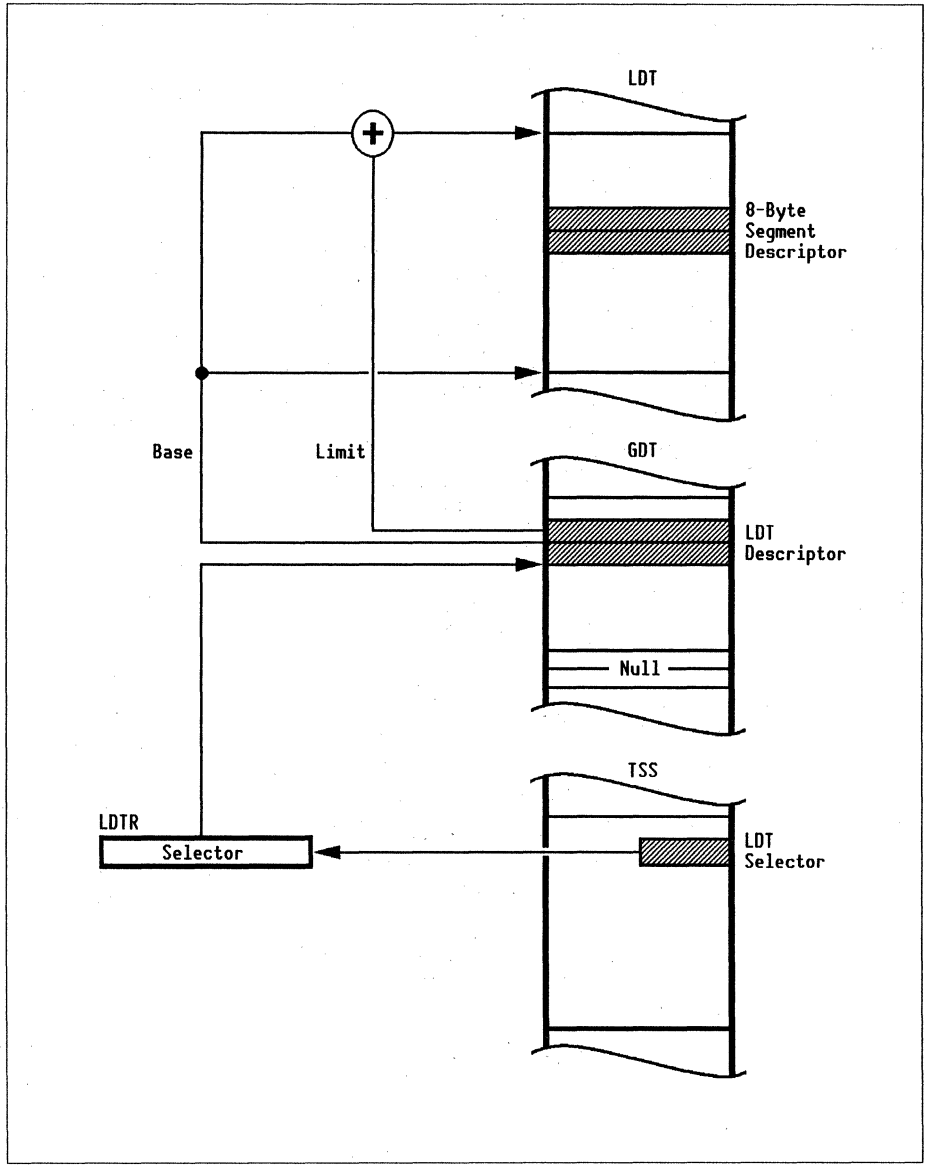
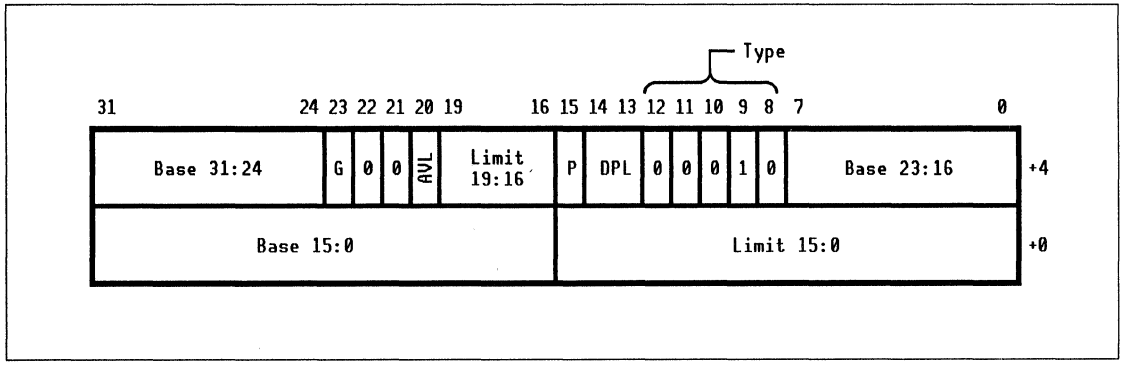


Figure 4-14 shows the format of an LDT descriptor. The LDT descriptor is loaded automatically into the invisible 64-bit LDT shadow register when the LDTR selector is loaded.

Figure 4-14. *Local Descriptor Table (LDT) Descriptor*



(+4 is high dword, +0 is low dword)

31:24 7:0 31:16	(+4) (+4) (+0)	Base	Segment Base Adress—These bits represent the 32-bit linear address of the segment's base in memory.
23	(+4)	G	Granularity—The G bit determines the maximum segment size (limit): 0 Byte-granular limit; the maximum segment size is 2^{20} bytes. 1 Page-granular limit; the maximum segment size is 2^{32} bytes. When the G-bit is set to 0, the 20-bit limit value, limit 19:0, is zero-extended to 32-bits. This provides the byte-granular limit. When the G-bit is set to 1, the 20-bit limit value is shifted left by 12 bits and OR'd with 0FFFh, thus providing a 32-bit limit value that is page-granular.
20	(+4)	AVL	Available to Software—This bit may be used by system software. It is not interpreted by the processor.
19:16 15:0	(+4) (+0)	Limit	Segment Limit—These bits indicate the 20-bit limit of the segment. The limit is expanded to 32 bits by interpreting the granularity (G) bit.

15	(+4)	P	Present—If set, this attribute indicates that the descriptor is present in memory and therefore valid. If this bit is clear, an attempt to access the segment causes an exception. 1 Present (valid) 0 Not present (invalid).
14:13	(+4)	DPL	Descriptor Privilege Level—These bits indicate the privilege level of the descriptor. The DPL is used by the processor to determine access rights to the segment pointed at by the descriptor. 11 Privilege level 3 (least privileged) 10 Privilege level 2 01 Privilege level 1 00 Privilege level 0 (most privileged).
12:8	(+4)	Type	Type—These bits indicate the type of descriptor—An LDT must have 00010 in this field.

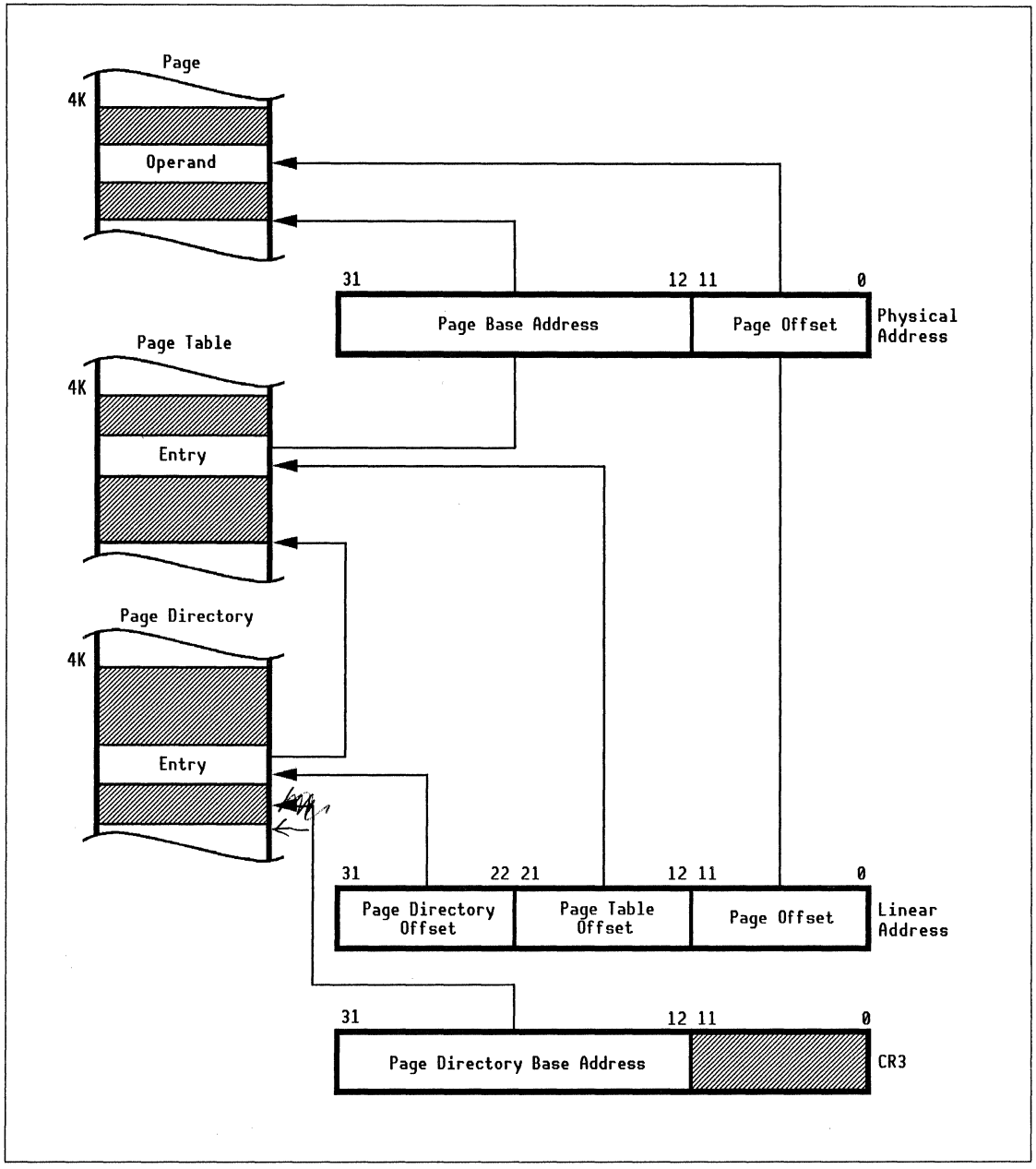
Paging

A page is a fixed 4kB block aligned to a 4kB boundary in physical memory. Paging translates the linear address provided by the segmentation system into physical pages. It does this by using a two-level arrangement of page directories and page tables. Figure 4-15 shows the paging mechanism.

Paging is enabled in protected mode when the PG bit in register CR0 is set to 1. The operating system normally keeps the segments relevant to its current task in memory. When a segmented linear address is translated to a physical address that is not in memory (as indicated by the present bit in either the corresponding page directory entry or page table entry), a page-fault exception is generated. The operating system's handler then reads the page from disk into memory, sets the present bit, and returns control. The system restarts at the instruction that generated the page-fault exception, and the program continues.

The CR3 register contains the base address for the current page directory, which must always be kept in physical memory. Register CR3 is changed during a task switch to accommodate tasks with different page directories.

Figure 4-15. *Paging Mechanism*



Entries in the Page Directories and Page Tables

The dword entries in the page directories and page tables have identical formats, except that one bit is unused in page directory entries. Figure 4-16 shows the format. Each page directory and page table can contain up to 2^{10} four-byte entries, each of which has the following fields:

- Base address for a page table or page.
- Dirty (D) bit—Indicates whether a page referenced by a page table entry has been written.
- Accessed (A) bit—Indicates whether a page or page table referenced by a page directory entry or page table entry has been read or written.
- User/Supervisor (U/S) bit—Indicates the privilege level required for access to a page table or page.
- Read/Write (R/W) bit—Indicates the read/write privilege for the user level.
- Present (P) bit—Indicates whether the table is currently in memory.

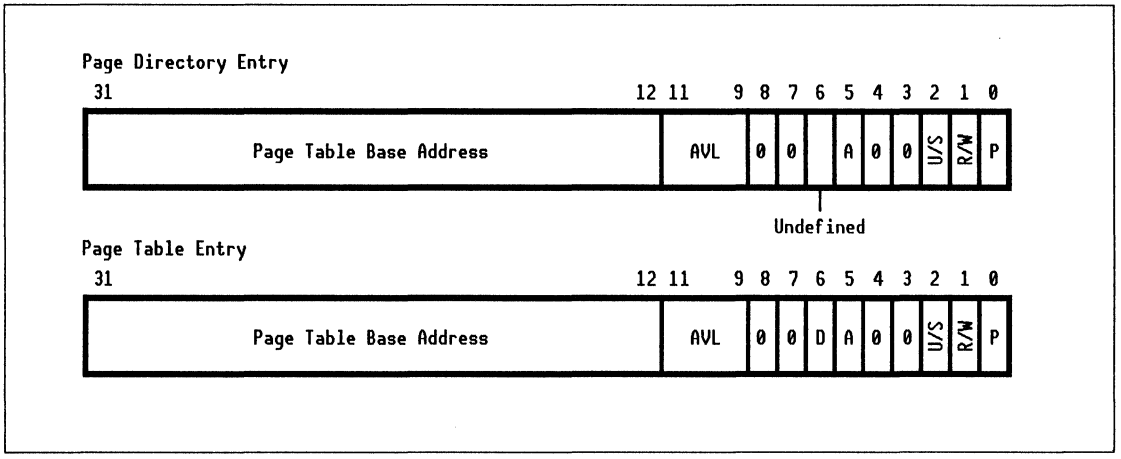
The processor sets the dirty bits and accessed bits, but it does not clear them. If the accessed bit is read and cleared periodically by the operating system, pages which have not been accessed since the last clearing of the bit can be identified and moved off to disk. If the dirty bit is cleared by the operating system before a page table or page is copied from disk to memory, the operating system will know whether the disk version needs to be updated when the page table or page is removed from memory.

Page tables and pages not in memory are identified by the present bit in their corresponding page directory entry or page table entry. The following minimum paging information must always be present in physical memory:

- Page directory pointing to the page-fault handling code
- Page table pointing to the page-fault handling code
- Page containing the page-fault handling code.

All other page directories, page tables, and pages can be left on disk and brought into memory as needed. When a page table or page is not present, the high-order 31 bits of its corresponding page directory or page table entry can be used by the operating system to store information, such as its location on disk. Figure 4-16 shows this format.

Figure 4-16. Format of Page Directory and Page Table Entries

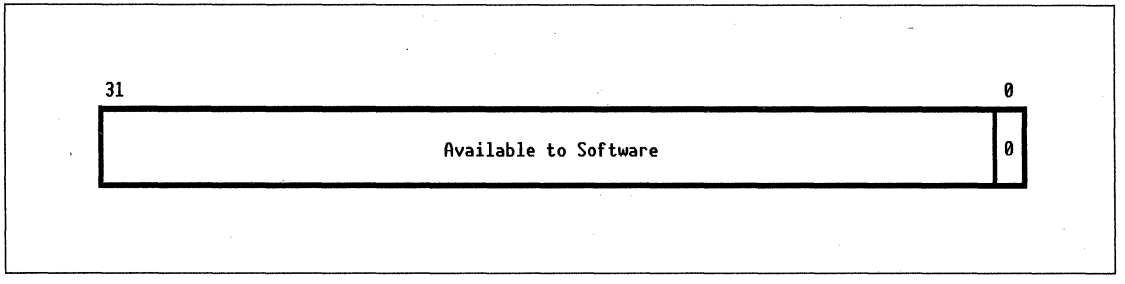


31:12	Base	Page Table or Page Base Address—These bits contain the 20-bit base address of the page table or page.
11:9	AVL	Available to Software—The three AVL bits are reserved for use by system software. They are not interpreted by the processor.
6	D	Dirty—The D bit is undefined in page directory entries. In page table entries, it is set to 1 by the processor during a write access to the page mapped by the page table entry. The D bit is never cleared by the processor, but it can be cleared by the operating system before the page is brought into memory to determine whether a write-back to disk is necessary during page swapping. 1 Dirty (write to page occurred) 0 Clean (no write to page).
5	A	Accessed—In page directory or page table entries, the A bit is set to 1 by the processor during a read or write access to the page table or page mapped by the entry. The A bit is never cleared by the processor, but it can be cleared by the operating system to obtain page table and page usage data. 1 Accessed (read or write) 0 Not accessed.

2	U/S	User/Supervisor—This bit is the maximum CPL that a code segment can have to access the page table or page mapped by the page directory or page table entry. The U/S bit in a page directory entry applies to all page tables (and associated pages) mapped by that entry. 1 User (privilege level 3) 0 Supervisor (privilege level 0, 1, or 2).
1	R/W	Read/Write—For the user privilege level (U/S = 1), this bit indicates whether pages mapped by the page directory or page table entry are read-only or read/write. The bit is not interpreted for supervisor level. 1 Read or write 0 Read only.
0	P	Present—The P bit indicates that the page table or page mapped by the entry is present in memory. It is set and cleared by the operating system. The current page directory must always be present in physical memory, but the other page directories and the page tables (except the one containing the entry for the page-fault handler code) can be not-present. If not present, bits 31:1 of the entry can be used by the operating system to store information, such as the location on disk of the page table or page. See Figure 4-17 for the not-present entry format. 1 Present in memory 0 Not present in memory.

Does dos do this?

Figure 4-17. *Format of Not-Present Entries (Page Directory or Page Table)*



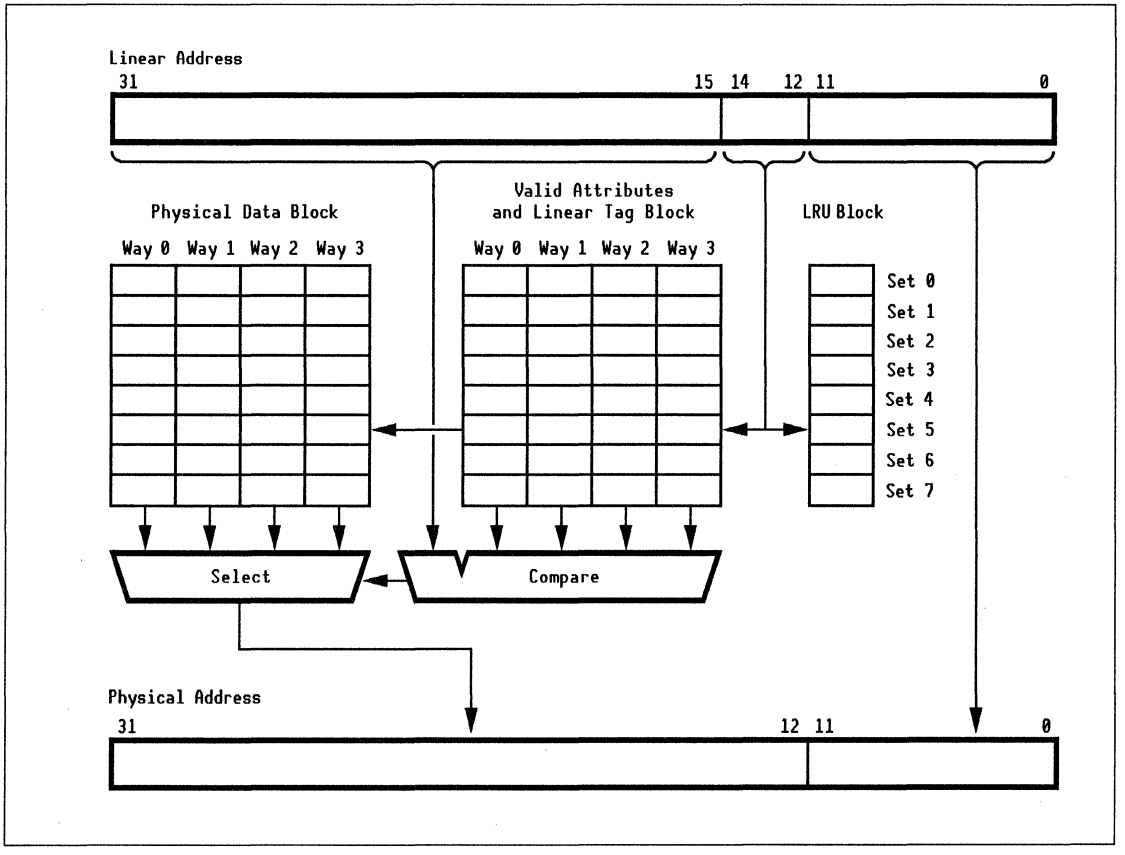
Translation Lookaside Buffer (TLB)

Put this before the discussion on p 4-30,31

The translation lookaside buffer (TLB) is an on-chip cache used by the processor to store essential parts of the most recently used page directory and page table entries (Figure 4-18). The processor reaches most accessed pages by using these entries in the TLB. If the referenced page cannot be found using the TLB (called a TLB *miss*), the processor attempts to create a translation using the page directory/page table lookup mechanism shown in Figure 4-15.

Updating of the TLB from the page directory/page table translations available in memory can take between 6 and 16 clocks, depending on the bits that need to be updated. If the present bit is cleared in the relevant page directory entry, indicating that the page table and page are not present in memory, or if the operation would violate the settings of the U/S and R/W bits in either the page directory or page table entry, a page-fault exception is generated.

Figure 4-18. *Translation Lookaside Buffer*



The page fault handler can then read the page from disk into memory, set the present bit, and return control. The system restarts at the instruction that generated the page-fault exception, and the program continues.

The processor does not maintain coherence between the TLB entries and the corresponding versions in memory. The operating system must therefore flush the TLB after any software modification of the page tables. This is done by moving the content of register CR3 to a general register and then moving it back again. For example,

```
MOV EAX, CR3 ; Move CR3 value to EAX
MOV CR3, EAX ; And move it back to flush the cache
```

During a task switch, in which the new task has a different page directory than the current task, the processor automatically updates the CR3 register with the stored CR3 value in the TSS and flushes the page table entries in the TLB.

The processor has a special set of registers for testing the TLB page translations. The section entitled "Testing the TLB" describes the mechanism.

Page Aliases

There are no restrictions on page aliasing. Translation tables can be constructed to cause multiple linear addresses to map to a single physical page. When this is done, however, multiple translation paths lead to a single physical page, complicating the use of the accessed and dirty bits in the tables. Because this information is somewhat linear-address dependent, it is necessary to examine all the translation entries for each linear address range to determine whether a physical page has been altered or referenced.

It is also possible to support inconsistent levels of protection. Two linear address ranges can map to the same physical address. One range may provide a different kind of protection than another. An operating system that determines which pages to deallocate must be aware of all the aliases by which each physical page can be accessed.

Paging and Multiprocessing

In a system with multiple processors, special care must be taken if a program executing on one processor modifies a page table that may be accessed simultaneously by a second processor. The Super386 processor supports this configuration by using indivisible read/modify/write cycles whenever it updates a page table entry to set the D or A bit.

Software updates to the page table will work properly if the LOCK prefix is used with instructions that modify the page table. Before changing a page table entry that may be used by another processor, software should use a locked AND instruction to clear the P bit in an indivisible operation. Then the entry can be changed as required, and made available by setting the P bit to 1.

At some point in the modification of a page table entry, all processors in the system that may have the entry cached must be notified (usually with an interrupt) to flush their TLBs. Until these old copies are flushed, these processors continue to access the old page, and may also set the D bit in the entry being modified. If this causes the modification of the entry to fail, the paging caches should be flushed after the entry is marked not present, but before the entry is otherwise modified.

Control Gates and System Calls

Control gates are descriptors. They are available only in protected mode and are used in system calls or traps, task switches, and interrupts and exceptions. Unlike segment descriptors, which point to a segment directly, control gates point to another descriptor (a segment descriptor), which locates the destination segment. They are an indirect means of transferring execution control to other code segments at the same level or a more privileged level. The indirection provides an opportunity for the processor to thoroughly check attributes and access rights, switch stacks (call gates to a more privileged level), and switch tasks (task gates).

There are four types of gate descriptors:

- Call gates
- Task gates
- Interrupt gates
- Trap gates.

These are described in the following paragraphs.

Call Gates—Call gates facilitate inter-procedure calls and jumps. Calls can be made to more privileged levels and are commonly used for system calls. Call gates can reside in the LDT or GDT and can pass parameters.

Task Gates—Task gates implement task switching and can reside in the GDT, LDT, or IDT. Task gates point to a TSS descriptor, which in turn points to a TSS.

Interrupt Gates—Interrupt gates facilitate access to interrupt handlers (service routines). They reside only in the IDT and can be invoked with the INT *n* instruction.

Trap Gates—Trap gates are identical to interrupt gates, except that the interrupt flag (IF) in the EFLAGS register is not cleared. Like interrupt gates, they can reside only in the IDT. ie. INTR signal is still recognized.

Figure 4-19 shows how control gates work. Whereas segment descriptors contain the base address and limit of a segment, control gates contain a selector and (except for task gates) an offset. The selector points to a segment descriptor, which points to a segment. The control gate's offset indexes into the segment.

Figure 4-19. *Control Gate Mechanism*

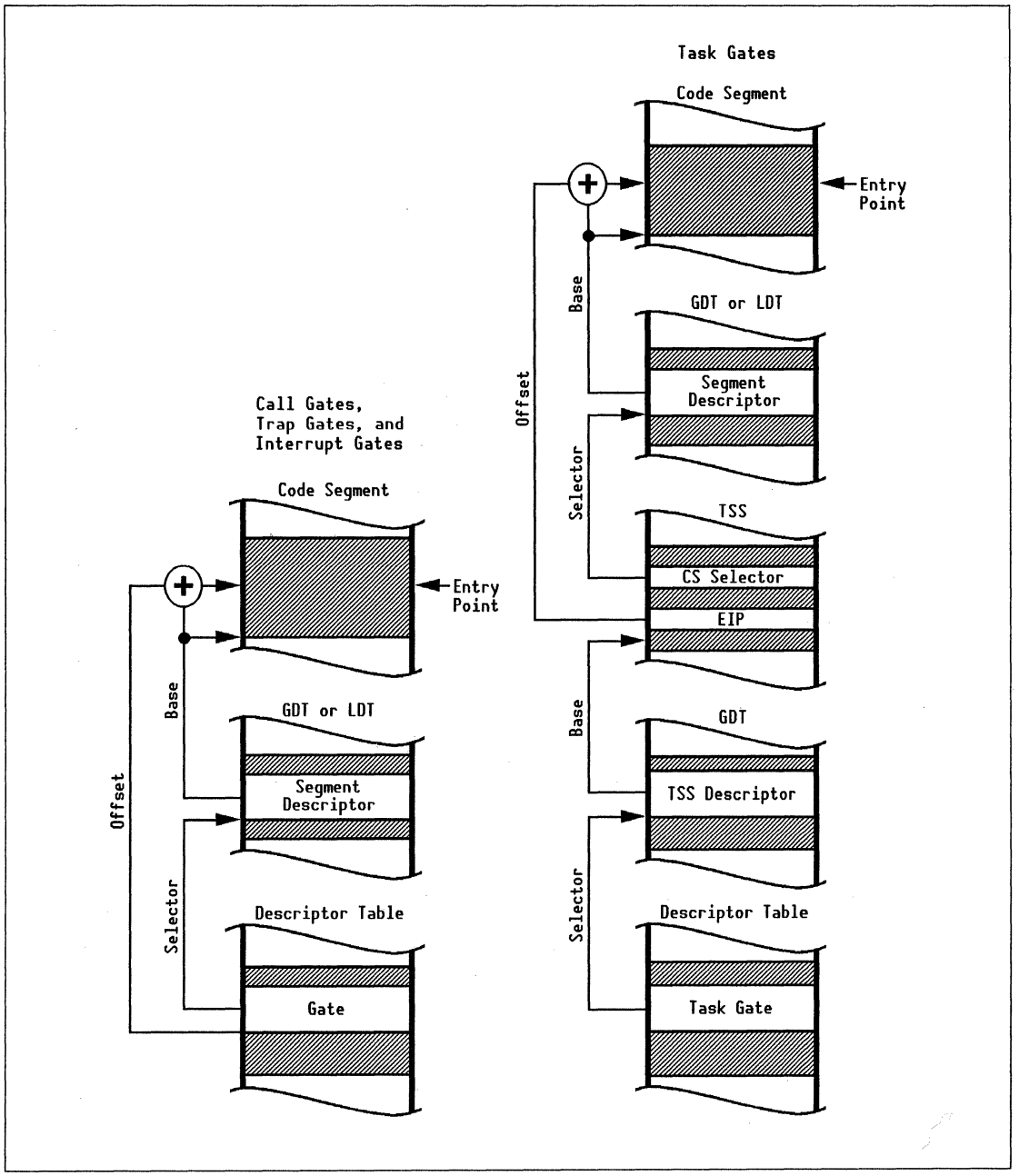


Table 4-4 compares the four types of gates. The notation used in this table for privilege level is defined in the section entitled "Protection Mechanisms."

Table 4-4. *The Four Types of Gate Descriptors*

	Call Gate	Task Gate	Interrupt Gate	Trap Gate
Purpose	Inter-segment jumps and system calls	Inter-task jumps and system calls	Interrupts, exceptions, and system calls	Interrupts, exceptions, and system calls
Location	GDT or LDT	GDT, LDT, or IDT	IDT	IDT
Passes parameters?	Yes	No	No	No
CALL-instruction rule checking and actions	$DPL_{gate} \geq \max(CPL, RPL_{gate})$ and $DPL_{code} \leq CPL$ and If $DPL_{code} < CPL$, do stack switch using SS and ESP in TSS, and copy call parameters.	$DPL_{gate} \geq \max(CPL, RPL_{gate})$ and Do task switch.	Not available	Not available
JMP-instruction rule checking and actions	$DPL_{gate} \geq \max(CPL, RPL_{gate})$ and $DPL_{code} \leq CPL$	$DPL_{gate} \geq \max(CPL, RPL_{gate})$ and Do task switch.	Not available	Not available
INT-*n*-instruction rule checking and actions	Not available	$DPL_{gate} \geq CPL$	$DPL_{gate} \geq CPL$ and Clear IF flag.	$DPL_{gate} \geq CPL$

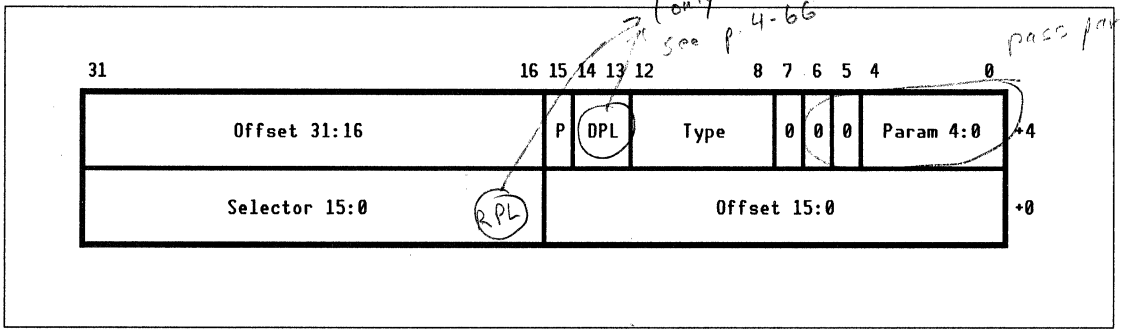
The descriptor formats for all gates, as well as their general functions, are described in the following section, "Control Gate Descriptors." The call gate mechanism is discussed in the section "Call Gates." Task gates are described further in "Multitasking," and interrupt gates and trap gates are discussed in "Interrupts and Exceptions."

Control Gate Descriptors

All four types of gates have a similar format, shown in Figure 4-20, although not all fields are used by all gate types. All gates use the present bit, DPL, and code segment selector. All but task gates also use the offset (entry point) into the code segment. The parameter (dword count) field is only used for call gates. This field contains the number of dword parameters to copy from the calling procedure's stack to the called procedure's stack.

Figure 4-52 in the section entitled "Interrupts and Exceptions" gives somewhat more detailed images of the fields used by interrupt, trap, and task gates.

Figure 4-20. *Control Gate Descriptor*



(+4 is high dword, +0 is low dword)

31:16 15:0	(+4) (+0)	Offset	Offset—The offset is the entry point index into the destination procedure. It is added to the base address of the destination procedure's code segment, obtained from the segment's descriptor, to determine the entry point. The offset (operand) in a call instruction is ignored. In a task gate, this field is not used.
15	(+4)	P	Present—If set, this attribute indicates that the gate descriptor is present in memory and is therefore valid. If this bit is clear, an attempt to access the gate causes an exception. 1 Present (valid) 0 Not present (invalid).

14:13	(+4)	DPL	Descriptor Privilege Level—DPL gives the privilege level of the descriptor. The DPL is used by the processor to determine access rights to the segment that the descriptor points to. 11 Privilege level 3 (least privileged) 10 Privilege level 2 01 Privilege level 1 00 Privilege level 0 (most privileged).
12:8	(+4)	Type	Type—The type field indicates the type of gate descriptor: 00100 Call gate (16-bit) 00101 Task gate 00110 Interrupt gate (16-bit) 00111 Trap gate (16-bit) 01100 Call gate (32-bit) 01110 Interrupt gate (32-bit) 01111 Trap gate (32-bit).
4:0	(+4)	Param	Parameter—This field is only used in call gates, where it specifies the number of doubleword parameters to copy from the caller's stack to the called procedure's stack.
31:16	(+0)	Selector	Segment Selector—These bits select the descriptor for the destination segment. In call gates, interrupt gates, and trap gates, they select a code segment descriptor in the GDT or LDT. In a task gate, they select a TSS descriptor in the GDT.

Call Gate

Call gates implement calls and jumps to code at the same or more privileged levels (lower privilege numbers). Only CALL instructions can use gates to transfer to more privileged levels. JMP instructions only use a gate to transfer control to a code segment with the same privilege level or to a conforming code segment.

Call gates are the only control gates that can pass parameters and switch stacks without switching an entire task. During a call, the processor checks the DPL of the call gate. The call is executed only if the DPL is greater (less privileged) than both the CPL and the RPL of the selector contained in the the call gate. If a more privileged segment is being called, a new stack is created using the stack segment and stack pointer for that privilege level, contained in the current TSS. (See the section entitled "Multitasking" for more on task state segments.)

Protection Mechanisms

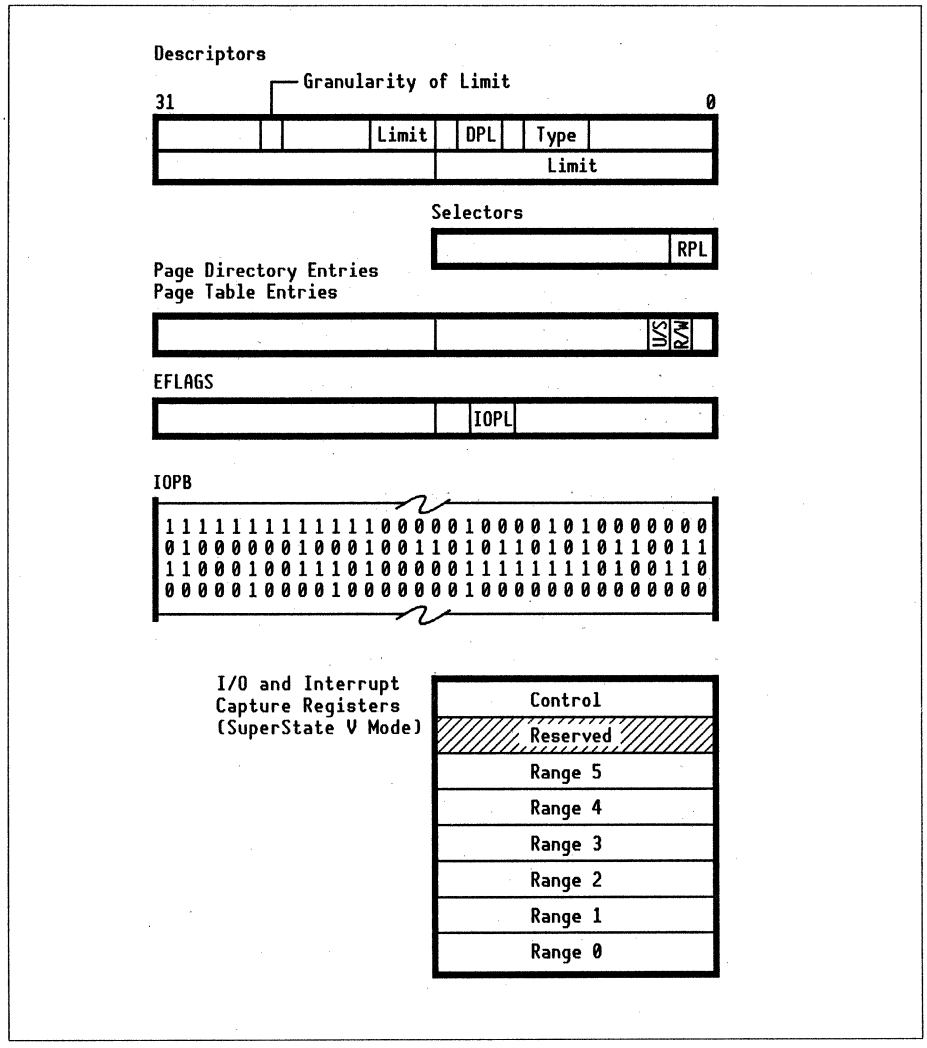
The processor provides protection mechanisms at the following levels:

- Segment descriptor
- Control gate descriptor
- Page
- I/O.

At the segment and gate descriptor levels, bit fields are used to control access by segment type, limit, and access privilege. At the page level, the page directories and page tables can be used to control read or write access by privilege level. Access to I/O resources can be controlled on the basis of global privilege level or on a port-by-port basis. In addition, the SuperState V extensions provide a capture mechanism that is transparent to existing operating systems and allows system software to monitor and intercept specific interrupts and I/O accesses.

Figure 4-21 shows where these fields are stored. All of these mechanisms are under system software control. The section entitled "Summary of Privilege-Level Checking and the CPL" lists the checking rules that are associated with privilege level. Other checking rules are enforced by the processor in a manner consistent with the setting of their related control fields. The section entitled "SuperState V Mode" describes the processor's power management and device virtualization functions.

Figure 4-21. *Data Structures Containing Privilege-Level Variables*



Compare Intel diagrams

Segment-Level Protection

Several fields in the segment descriptor and one field in the segment selector control access by procedures and tasks to the system resources. The descriptor fields include the segment type, its limit, and the descriptor's privilege level. The selector field contains a requestor privilege level, which is set by the operating system as required for software access or protection.

Type

Bits 12:8 of the segment descriptor are often referred to as the *type field.* These bits specify whether the segment is available to applications or the system, whether it is code or data, how the segment is sized and how it expands, and its read/write access privilege. These fields are written by the operating system at initialization and any other time thereafter. They are compared with the processor's access rules whenever a segment is accessed.

Limit

The segment limit is specified in the segment's descriptor. All ~~operand~~ accesses to the segment are checked against this limit. In expand-up segments, accesses *must not* exceed the limit. In expand-down segments, accesses *must* exceed the limit.

Privilege Level

All descriptors, including descriptors for LDTs, contain a field specifying the DPL. Privilege level 0 is the most privileged; level 3 is the least privileged. The operating system uses privilege levels to protect shared resources and functions among tasks. The operating system kernel is typically assigned privilege level 0. The processor checks the privilege level of segment selectors and segment descriptors during segment loading, control transfers, and task switches. I/O accesses use a seperate protection mechanism, namely, IOPL and IOPB. In most cases, the DPL of the code segment determines access, because code segments contain the instructions that could cause harm to system resources. Several variables related to privilege level are used. One of them, the CPL, is determined by the processor. The others are determined by the operating system.

The variables associated with segment selectors and descriptors are

- Current privilege level
- Descriptor privilege level
- Requestor privilege level.

These are discussed in the following paragraphs.

Current Privilege Level—The CPL is the only privilege variable that is determined by the processor. It is stored automatically in the RPL field of the code segment selector register after the code segment has been privilege-checked and loaded. The processor always considers the stack RPL to be equal to the CPL.

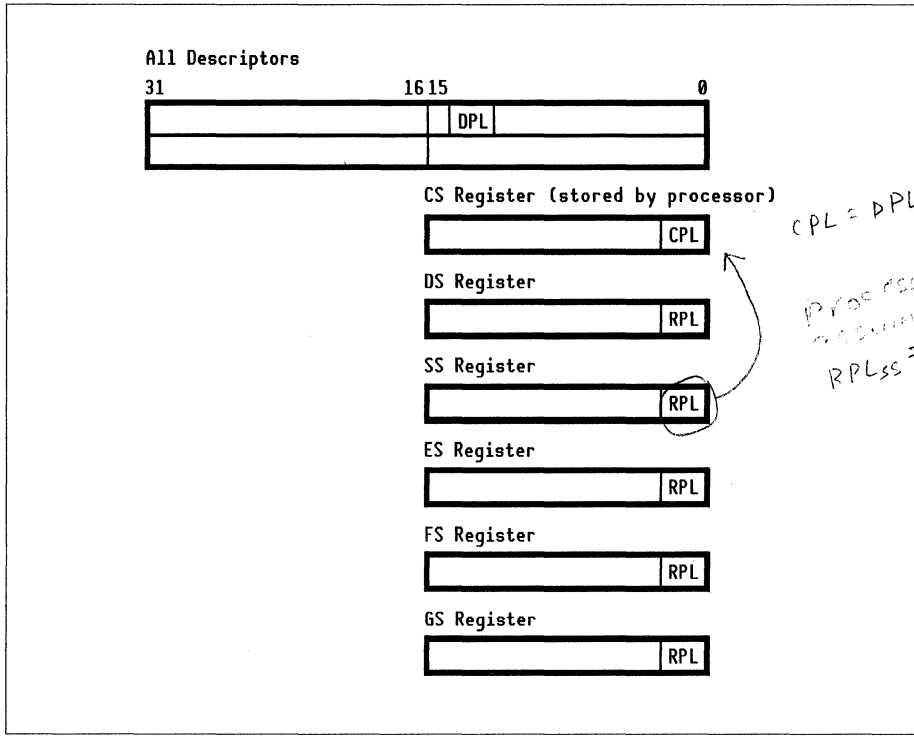
Descriptor Privilege Level—the DPL is the basic privilege level of a segment. It is checked whenever a segment selector is loaded into a segment register.

Requestor Privilege Level—The RPL is an override privilege level for a segment. It is checked whenever a segment selector is loaded into a segment register. The processor always considers the stack RPL to be equal to the CPL.

Figure 4-22 shows where the CPL, DPL, and RPL fields are stored. The section entitled "Summary of Privilege-Level Checking and the CPL" and Table 4-5 in that section contain the details of rule checking. When executing in SuperState V mode, the CPL is 0 and the processor can use SuperState V instructions and facilities, including the SuperState V memory and capture facility. For details, see the section entitled "SuperState V Mode."

In task switches and returns from far calls or interrupts, the RPL becomes the new CPL (see p. 4-52)

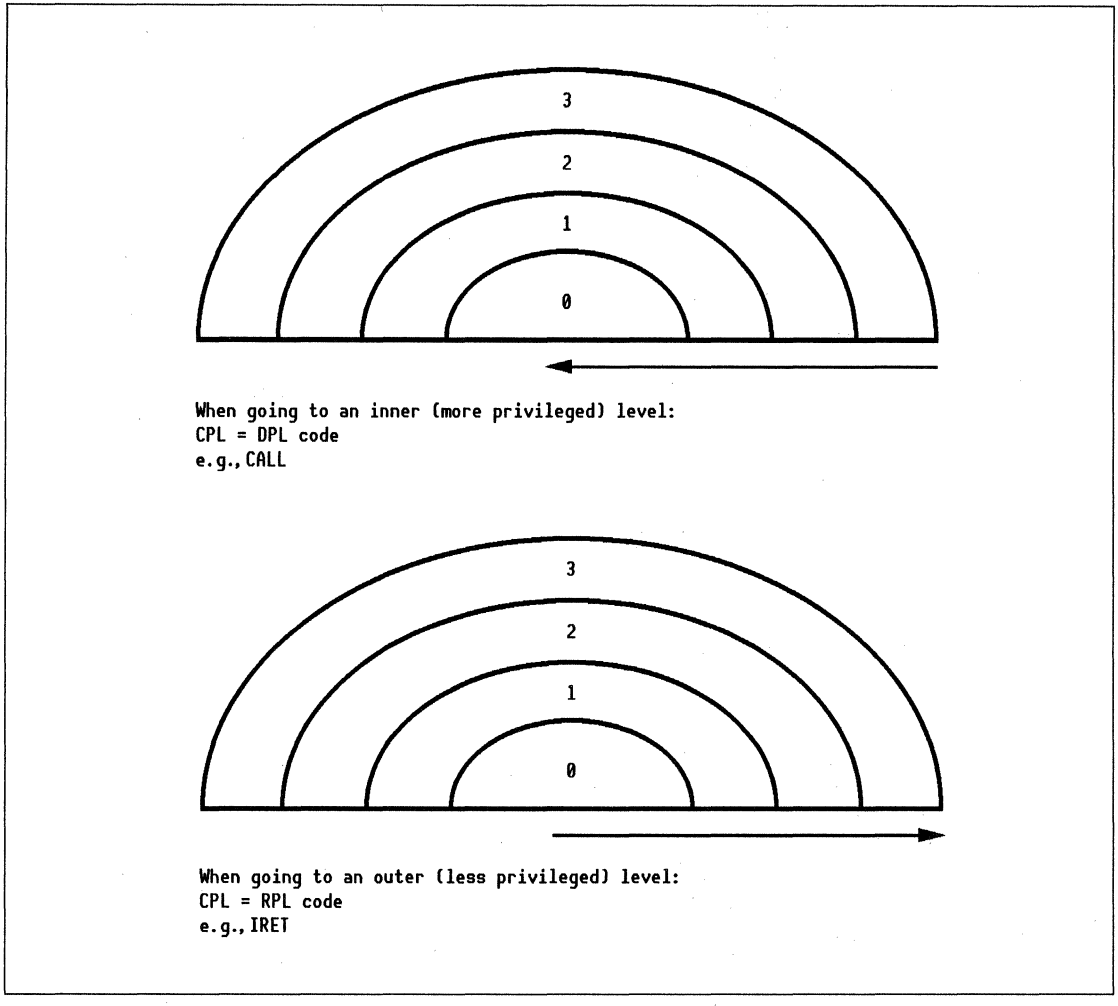
Figure 4-22. CPL, DPL, and RPL Fields



In most cases, an instruction may load a segment if the DPL of the segment is equal to or greater (less privileged) than the CPL and the RPL. ~~During segment loads, if the RPL is greater (less privileged) than the CPL, the RPL overrides the CPL.~~

In general, the processor sets the CPL equal to the DPL of the current code segment after checking the segment for privilege and loading it. The term CPL can therefore be considered an acronym for code privilege level, as well as current privilege level. For example, Figure 4-23 shows how the CPL is assigned for nonconforming code segments after a segment load that involves a change in privilege level. In this example, a CALL is made to an inner privilege level, and the CPL is taken from the DPL of the destination code segment. Upon return, the CPL is taken from the RPL of the code selector for the code segment to which control returns.

Figure 4-23. CPL Assignment for Nonconforming Code Segments



Attempts to access a segment with improper privilege generates an exception. Conforming code segments, however, can be accessed from less privileged levels. These segments are typically used for shared libraries and interrupt handlers. Refer to the section entitled "Conforming Code Segments."

Control Gate Protection

Control gates are descriptors that point to segment descriptors rather than directly to segments. There are four types of control gates: task gates, interrupt gates, trap gates, and call gates. Control gates implement transfers to code at the same or more privileged levels, or to different tasks. Like segment descriptors, gate descriptors have DPL and type fields that provide protection.

Control transfers are done with the jump, call, and return instructions, or by interrupts and exceptions. Near jumps, calls, and returns receive only limit checking. Far jumps, calls, and returns receive privilege-level checking. The rules vary, depending on the type of transfer and the type of gate. The section entitled "Summary of Privilege-Level Checking and the CPL" explains the rules.

Privilege level switching through call, interrupt, or trap gates also provides protection of stacks. Each privilege level has its own stack. When a program switches to a new CPL, the program creates a new stack at the new CPL using the stack pointer and stack segment selector stored in the TSS.

For more on control gates, see the section entitled "Control Gates and System Calls."

Page-Level Protection

When paging is enabled, the operating system checks the following fields in each entry of the page directories and the page tables:

- User/Supervisor CPL (U/S)
- Read/Write Access (R/W).

User level is privilege level 3; supervisor level is privilege level 0, 1, or 2. The U/S field specifies the maximum CPL that can access the directory or table. The operating system writes both U/S and R/W fields into each page directory entry, and page table entry, and they are checked whenever a page directory or page table is accessed.

When U/S privilege is combined with read and write access, the most restrictive attributes from either level apply. The values in a page directory entry take precedence over those in a page table. For example, if a page directory entry says a page is read-only but the page table entry says read/write, the page is read-only.

I/O Protection

When the I/O space is used instead of memory-mapped I/O, it has two levels of protection:

- I/O Privilege Level (IOPL)—Two-bit field in the EFLAGS register that is compared with the CPL.
- I/O Permission Bitmap (IOPB)—Data structure in each task's TSS that grants access on a port-by-port basis.

In protected mode, global protection is first applied through the IOPL. This specifies the maximum CPL required to execute I/O instructions. If the CPL ≤ IOPL test fails, I/O port-level protection is optionally provided by the IOPB in the TSS.

The IOPB contains access-control bits for individual bytes (ports) in the I/O space. To gain access to an I/O port, the executing code must have a CPL less than or equal to the IOPL, and, if an IOPB is used, the bit mapped to that I/O port must be cleared to 0. The mechanisms are described in more detail in the section entitled "I/O."

incomplete

IOPL also determines the maximum CPL allowed to alter the interrupt flag (IF). POPF and IRET instructions can alter the IOPL when they are executed from privilege level 0. A task switch always alters the IOPL when the new image of the flags is loaded from the new TSS.

Summary of Privilege-Level Checking and the CPL

Table 4-5 summarizes the processor's privilege-level checking and CPL setting. The following notation is used in the table:

CPL	Current privilege level. It is determined by the processor and stored automatically in the RPL field of the CS selector register after the code segment has been loaded. The processor always considers the stack RPL to be equal to the CPL.
DPL_{code}	Descriptor privilege level in a destination code segment's descriptor. It is checked whenever the segment selector is loaded.
DPL_{data}	DPL in a destination data segment's descriptor.
DPL_{gate}	DPL of a gate descriptor.
DPL_{TSS}	DPL of a task state segment descriptor. (source or destination?)
RPL_{code}	Requestor (override) privilege level of a destination code segment's selector.
RPL_{data}	RPL of destination data segment's selector. The processor always considers the stack RPL to be equal to the CPL.
RPL_{gate}	RPL of the selector contained in a gate descriptor.
RPL_{TSS}	RPL contained in the selector operand of an instruction that causes a task switch. This RPL is finally stored in the TR register after the instruction executes.
$U/S_{directory}$	User/supervisor field of a page directory entry.
U/S_{table}	User/supervisor field of a page table entry.
$R/W_{directory}$	Read/write field of a page directory entry.
R/W_{table}	Read/write field of a page table entry.
IOPL	I/O privilege level in the EFLAGS register.
IOPB	I/O permission bitmap in a task's TSS.

RPL instruction

Where does the processor look to find the selector of a destination segment descriptor which it must examine for the RPL? Answer: the instruction must provide it. (Yes?)

Table 4-5. *Privilege-Level Rules for Access or Control Transfer*

Segment or Function	Access Type	Privilege-Level Check (True = Pass)	CPL After Action
Data Segment	All	$DPL_{data} \geq max(CPL, RPL_{data})$	No change
Stack Segment	All	$DPL_{data} = CPL = RPL_{data}$	No change
Code Segment (Conforming)	All	$DPL_{code \text{ or } gate} \leq CPL$	No change
Code Segment (Non-Conforming)	Near jump or call	None	No change
	Far jump or call (no gate)	$DPL_{code} = CPL$ and $DPL_{code} \geq RPL_{code}$	$CPL = DPL_{code}$
	Far jump (call gate)	$DPL_{gate} \geq max(CPL, RPL_{gate})$ and $DPL_{code} = CPL$	$CPL = DPL_{code}$
	Far call (call gate)	$DPL_{gate} \geq max(CPL, RPL_{gate})$ and $DPL_{code} \leq CPL$	$CPL = DPL_{code}$
	Interrupt or exception (software)	$DPL_{gate} \geq CPL$	$CPL = DPL_{code}$
	Interrupt or exception (hardware)	None	$CPL = DPL_{code}$
	Return from far call or interrupt	$RPL_{code} \geq CPL$	$CPL = RPL_{code}$
	Task switch (direct)	$DPL_{TSS} \geq max(CPL, RPL_{TSS})$	$CPL = RPL_{code}$
	Task switch (task gate)	$DPL_{gate} \geq max(CPL, RPL_{gate})$	$CPL = RPL_{code}$
Paging	All	$U/S = min(U/S_{directory}, U/S_{table})$ and $R/W = min(R/W_{directory}, R/W_{table})$	No change
I/O	All	$IOPB_{port} = 0$ and $CPL \leq IOPL$ (protected mode only)	No change
	Protected mode (32-bit tasks)	$CPL \leq IOPL$ or $IOPB_{port} = 0$	No change
	Protected mode (16-bit tasks)	$CPL \leq IOPL$	No change
	Real mode	$CPL \leq IOPL$ (always succeeds since CPL = 0)	No change
	Virtual-8086 mode	$IOBP_{port} = 0$	No change

Privileged Instructions

The instructions listed in Table 4-6 are reserved for code segments at the highest current privilege level (CPL = 0). These instructions will cause a general-protection exception if used at a less privileged level.

Table 4-6. Privileged Instructions

Mnemonic	Description
CLTS	Clear task-switched flag
HLT	Halt
LGDT	Load GDT register
LIDT	Load IDT register
LLDT	Load LDT selector and shadow descriptor register
LMSW	Load machine status word
LTR	Load TSS selector register and TSS shadow descriptor register
MOV CR *n*	Move to/from control register
MOV DR *n*	Move to/from debug register
MOV TR *n*	Move to/from test register

Conforming Code Segments

Conforming code segments are accessible from any less privileged level. They are used for such things as shared libraries and interrupt handlers. They can be created by setting the C/ED bit and the E bit to 1 in the segment descriptor. For control to be successfully transferred, the DPL of the conforming segment (or the gate used to access it) must be less than or equal to the CPL. That is:

DPL_{code} or gate $\leq$ CPL

Multitasking

In a multitasking environment, such as protected mode optionally provides, the execution of several programs is interleaved so that the processor appears to run all programs simultaneously. Programs that run in this manner are called *tasks*. The processor supports the execution of multiple tasks with a combination of instructions, registers, and task-switching data structures.

The interleaving of execution is accomplished by a *task switch*. During task-switching, the context of the current task is saved, a new context for the new task is loaded, and the memory segments for the new task are made active. There are typically two parts to the context information about a task: the machine state (i.e., the contents of essential registers), and the software state. In the Super386 processor, the machine state is saved automatically during a task switch in a memory data structure called a Task State Segment (TSS). The operating system may also use the task state segment to store information about the software state during a task switch.

And there's only one TSS per task.

Task switches are similar to procedure calls, except that they save more information about the processor's state. They do not, however, push the contents of saved registers on the stack, as procedures calls do; instead, they store this information in their TSS at the completion of the task. Because of this, tasks are not re-entrant as are procedures. Tasks cannot be called by other tasks if they are already running or waiting to run.

The prioritizing of tasks is implemented by the operating system. Within these constraints, software can request a task switch in one of the following ways:

- Far call or jump
- Interrupt or exception
- Interrupt return.

These procedures are discussed in the following paragraphs.

Far Call or Jump—A call or jump to a different segment is executed when the instruction supplies a segment selector that references either a TSS descriptor or a *task gate*, ~~which manages access to tasks on the basis of privilege level.~~

Interrupt or Exception—A task switch can be initiated during an interrupt or exception in which a handling routine is called through a task gate descriptor in the interrupt descriptor table (IDT).

Interrupt Return—A task switch can also occur when an IRET instruction is executed with the nested task (NT) flag in the EFLAGS register set to 1.

Registers and Data Structures

Task switching is supported by the following memory data structures and on-chip registers:

- Task state segment
- Task state segment descriptor
- Task register
- Task gates.

These registers are discussed in the following paragraphs.

Task State Segment (TSS)—A TSS is a memory data structure that stores the processor context and other information identifying a task. Each task has one TSS, which is updated during each task switch.

Task State Segment Descriptor—A TSS descriptor is a memory data structure that identifies the size and location of a task state segment and characterizes it on the basis of presence, availability, privilege level, and granularity. Each task has one such descriptor, only a few bits of which are updated during each task switch.

Task Register (TR)—A task register is a 16-bit visible register containing the TSS selector. The TR is accompanied by a 64-bit invisible shadow register that is loaded automatically with the TSS descriptor whenever the TR register is loaded. Together, via the global descriptor table, the TR and its shadow register locate the task state segment.

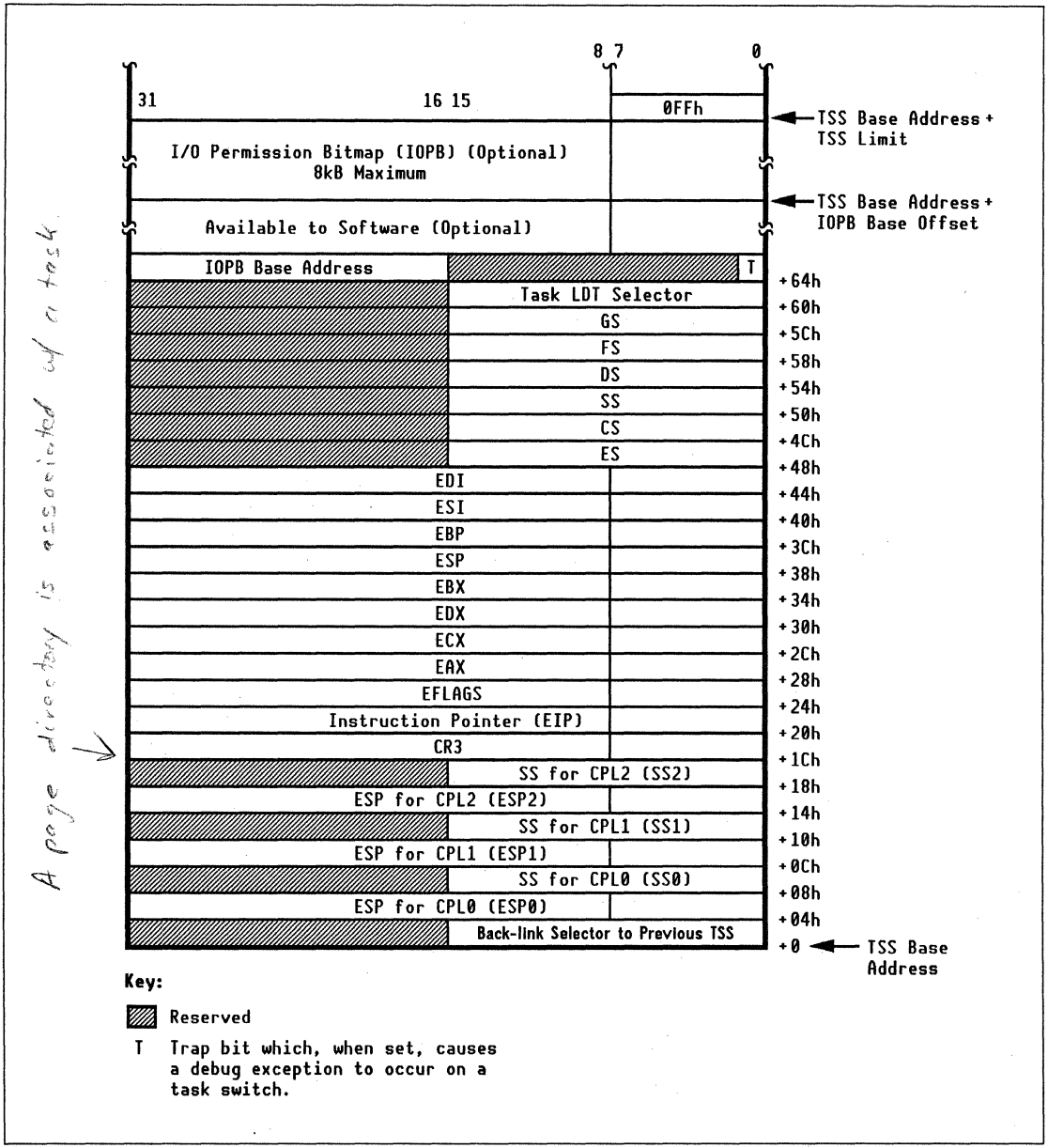
Task Gates—Task gates are memory structures in the global descriptor table, local descriptor tables, and/or interrupt descriptor table that manage access to TSS descriptors ~~based on privilege level.~~

These data structures and registers are described in more detail in the following sections, and they are summarized in Appendix B, "Super386 Quick Reference."

Task State Segment

Task state segments are data structures in memory. They must be at least 104 bytes in size and may be up to 64kB in size. They hold the machine state (essential register values) of a task as well as static information about the task. Each task has one task state segment. Its structure is shown in Figure 4-24. The machine state (dynamic fields) is updated automatically by the processor at every task switch. The static fields are initialized by the operating system during creation of the TSS. The processor reads and writes the TSS on task switches and reads it during changes in privilege level.

Figure 4-24. Task State Segment (TSS) Structure



The minimum size of the TSS is 104 bytes. If an I/O permission bitmap (IOPB) is used to protect access to I/O ports by privilege level, it must occupy addresses above the task state segment. In addition, the operating system may store other information between address 67h and the IOPB.

Among the dynamic fields is a back-link field in which the segment selector of the TSS descriptor for the previous task is stored. This allows an IRET instruction to restore the previous processor context and continue an interrupted task.

The IOPB base displacement is the offset from the base of the TSS to the base of the optional IOPB. The trap (T) bit can be set to cause a trap to the debug exception handler when a task switch occurs.

Fields are also provided in the TSS for three stack segment selectors and three stack pointers, which correspond to privilege levels 0, 1, and 2. These are used for privilege-level changes (from less privileged to more privileged) such as system calls, interrupts, or exceptions. When a privilege-level change occurs, the stack for the more privileged level is used. This is done by loading the more privileged SS and ESP registers from the TSS.

Static Fields

The static fields, read by the processor but not changed, are set up by the operating system when the task is created. They include:

- Stack segment selectors: SS2, SS1, and SS0
- Stack pointers: ESP2, ESP1, and ESP0
- Local descriptor table (LDT) selector
- Trap (T) bit
- IOBP base displacement
- Page directory base address: CR3.

These fields are discussed in the following paragraphs.

Stack Segment Selectors (SS2, SS1, and SS0)—Stack segment selectors for privilege levels 0, 1, and 2 must be initialized for all privilege levels that are used. They are loaded along with ESP2, ESP1, and ESP0 during system calls, interrupts and exceptions involving changes to a greater privilege level, which causes a stack switch.

Stack Pointers (ESP2, ESP1, and ESP0)—Stack pointers for privilege levels 0, 1, and 2. These must be initialized for all privilege levels that are used. They are loaded along with SS2, SS1, and SS0 during system calls, interrupts, and exceptions involving changes to a greater privilege level, which cause a stack switch.

Local Descriptor Table (LDT) Selector—The LDT field should be initialized to the task's LDT selector, or to a null selector if no LDT is used.

Trap (T) Bit—The trap bit is used for debugging. When set to 1, it causes a trap (exception 1) to the debug exception handler when a task switch to this task occurs. The breakpoint trap (BT) bit (bit 15) of the DR6 register indicates the trap condition.

IOPB Base Displacement—The IOBP base displacement locates the I/O permission bitmap, which contains one bit for every 8-bit I/O port. The map allows each task to protect each I/O port on the basis of the task's privilege level. This field must be initialized with the displacement of the IOPB from the base address of the TSS. For details, see the section entitled "I/O Permission Bitmap (IOPB)."

Page Directory Base Address (CR3)—If paging is enabled, the page directory base address field must be initialized with the physical address of the task's page directory.

Dynamic Fields

The dynamic fields of the task state segment, which the processor updates during each task switch, include:

- Back-link to previous TSS
- Instruction pointer and flags registers: EIP and EFLAGS
- General registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP
- Segment registers: CS, DS, SS, ES, FS, and GS.

Back-link to Previous TSS—The back link to the previous TSS is the segment selector of the TSS descriptor for the previous task. This field allows an IRET instruction to restore the previous task context so that nested, disjoint tasks can be run. See the section entitled "Nested (Linked) Tasks."

Instruction Pointer and Flags Registers—The EIP, EFLAGS, and general register fields should be initialized to values that the task needs when it begins execution.

General Registers—The EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP fields should be initialized to values that the task needs when it begins execution.

Segment Registers—The CS, DS, SS, ES, FS, and GS segment register fields should be initialized with selectors for their respective segments or with a null selector for those not used.

A TSS descriptor cannot be referenced through a segment selector, so the TSS cannot be initialized by writing directly to it. Instead, a data segment alias (synonym) must be used. This is a data segment that occupies the same linear addresses as the TSS, or that occupies pages which, via the paging mechanism, are mapped to the TSS pages. Figures 4-25 and 4-26 show how a TSS can be read and written using an alias descriptor in the segmentation and paging mechanisms, respectively.

Figure 4-25. *Accessing a TSS With a Segmentation Alias*

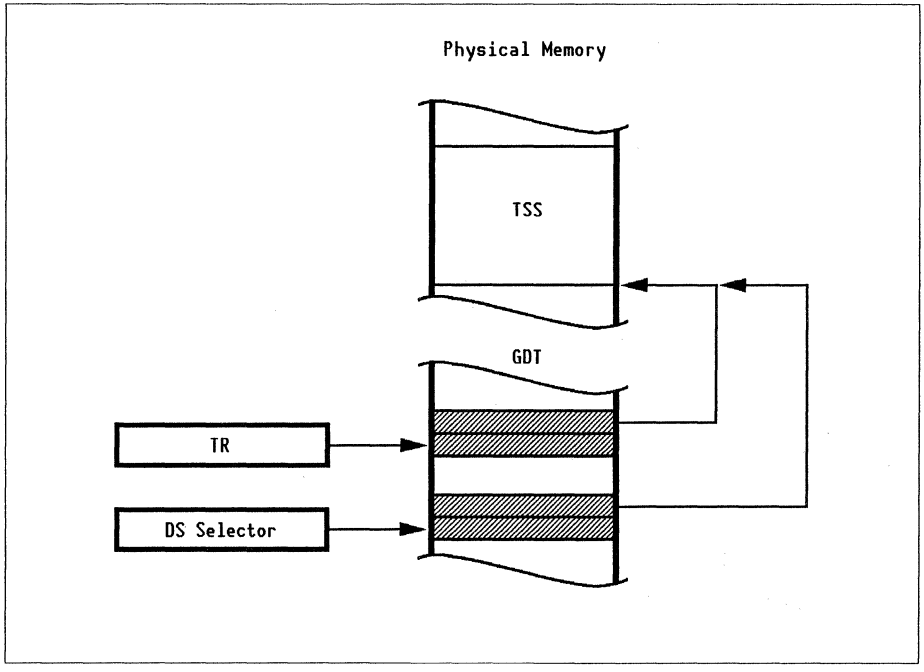
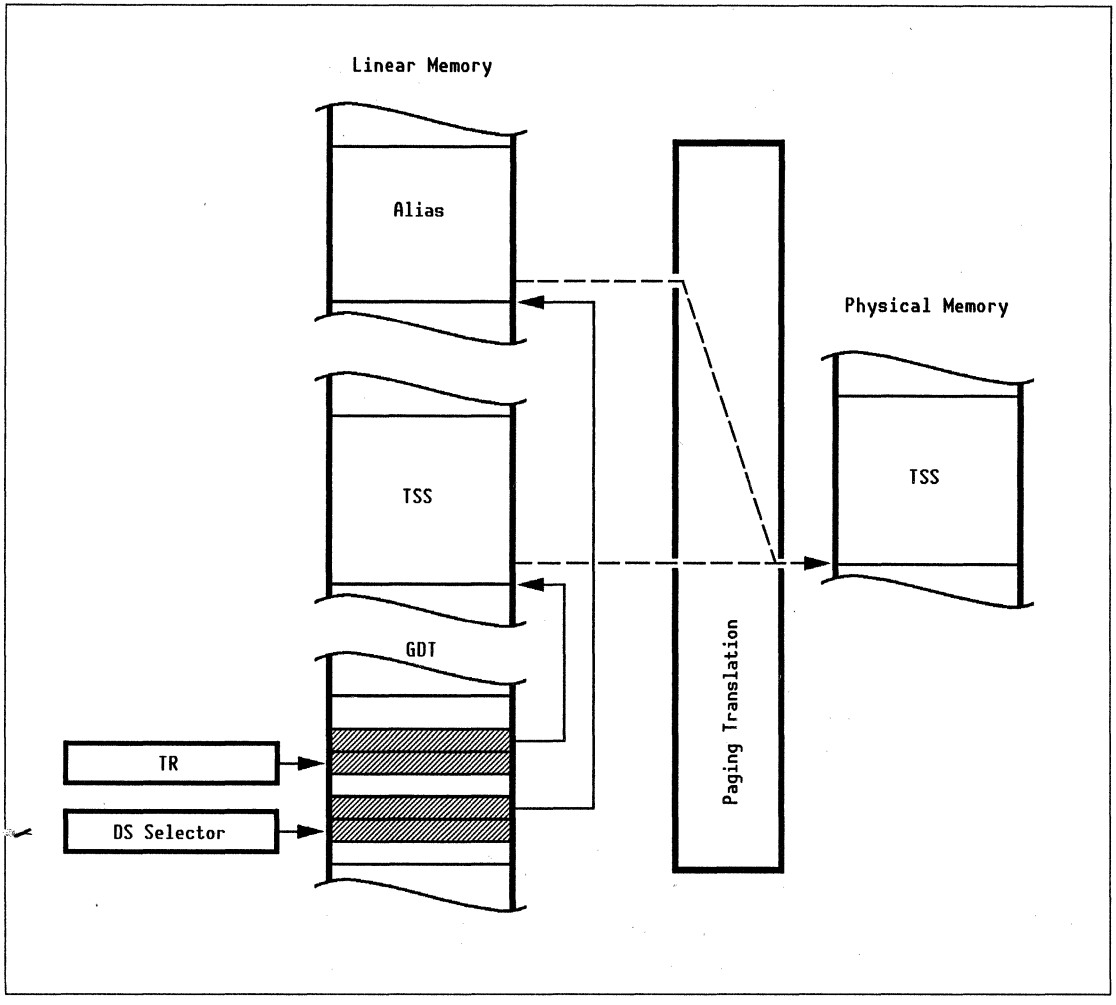


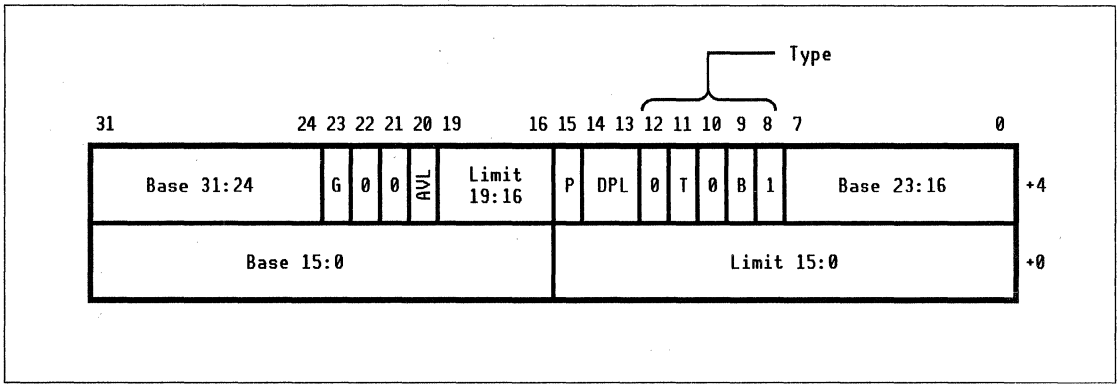
Figure 4-26. *Accessing a TSS With a Paging Alias*



TSS Descriptor

Each TSS has a descriptor that identifies the size and location of the segment and characterizes it on the basis of presence, availability, privilege level, and granularity. The descriptor is stored only in the GDT. A single bit is updated during each task switch. The format of the TSS descriptor is illustrated in Figure 4-27.

Figure 4-27. *TSS Descriptor*



(+4 is high dword, +0 is low dword)

31:24 7:0 31:16	(+4) (+4) (+0)	Base	Segment Base Address—The base bits are the parts of the 32-bit linear address of the segments base in memory.
23	(+4)	G	Granularity—The G bit determines the maximum segment size (limit): 0 Byte-granular limit; the maximum segment size is 2^{20} bytes. Page-granular limit; the maximum segment size is 2^{32} byte. When the G-bit is set to 0, the 20-bit limit value, limit 19:0, is zero-extended to 32-bits. This provides the byte-granular limit. When the G-bit is set to 1, the 20-bit limit value is shifted left by 12 bits and OR'd with 0FFFFh, thus providing a 32-bit limit value that is page-granular.

20	(+4)	AVL	Available to Software—This bit may be used by system software. It is not interpreted by the processor.
19:16 15:0	(+4) (+0)	Limit	Segment Limit—The two parts of the segment limit are expanded to 32 bits by interpreting the G bit. The limit must have a value of 67h or greater; it must always be greater than 67h if an IOPB is used.
15	(+4)	P	Present—If set, this attribute indicates that the descriptor is valid. If this bit is clear, an attempt to access the segment causes an exception. 1 Present (valid) 0 Not present (invalid).
14:13	(+4)	DPL	Descriptor Privilege Level—DPL indicates the privilege level of the descriptor. These bits determine the minimum privilege level needed to access the memory segment pointed at by the descriptor. 11 Privilege level 3 (lowest) 10 Privilege level 2 01 Privilege level 1 00 Privilege level 0 (highest).
12:8	(+4)		Bits 12:8 are sometimes referred to as the type field. The individual bits are specified explicitly in Figure 4-27 and in the T and B bits described below.
11	(+4)	T	TSS Type—This bit indicates the type of descriptor. 1 32-bit (Super386) TSS 0 16-bit (80286) TSS.
9	(+4)	B	Busy—This bit indicates whether the task is busy (running or waiting to run) or available: 1 Busy 0 Not busy (available).

The minimum limit of the TSS is 67h (104 bytes). This limit may be increased to account for additional space used by the operating system to store the state of software and/or the IOPB. If an IOPB is used, it must occupy addresses above the task state segment. In addition, the operating system may store other information between address 67h and the IOPB.

An indication of the task's execution status is encoded in the busy bit (bit 9) of the upper dword. This bit should be initialized to 0 by the operating system. The processor sets this bit to 1 when the task is run so as to trap re-entrant attempts to invoke the task. A general-protection exception is triggered if an attempt is made to call a busy task.

In non-nested task switching, the processor sets the busy bit of the new task and clears it in the old task. In nested task switching, the processor sets the busy bit of the new task and also leaves the old task's busy bit set, to prevent re-entrant task switching. When setting or clearing the busy bit, the processor locks the external bus. This prevents two processors in a multiprocessing environment from accessing the same task simultaneously. Table 4-7 shows the changes made by the processor to the busy bit, NT flag, and TSS back-link field for both the old and new tasks.

Table 4-7. *Processor Changes During Task Switch*

	Busy Bit (TSS Descriptor)		NT Flag (EFLAGS)		Back-link Field (TSS)	
	Old Task	New Task	Old Task	New Task	Old Task	New Task
Jumps	0	1	X	0	X	X
Call	X	1	X	1	X	old TSS selector
Interrupts or Exceptions	X	1	X	1	X	old TSS selector
Return from Interrupt	0	X	0	X	X	X

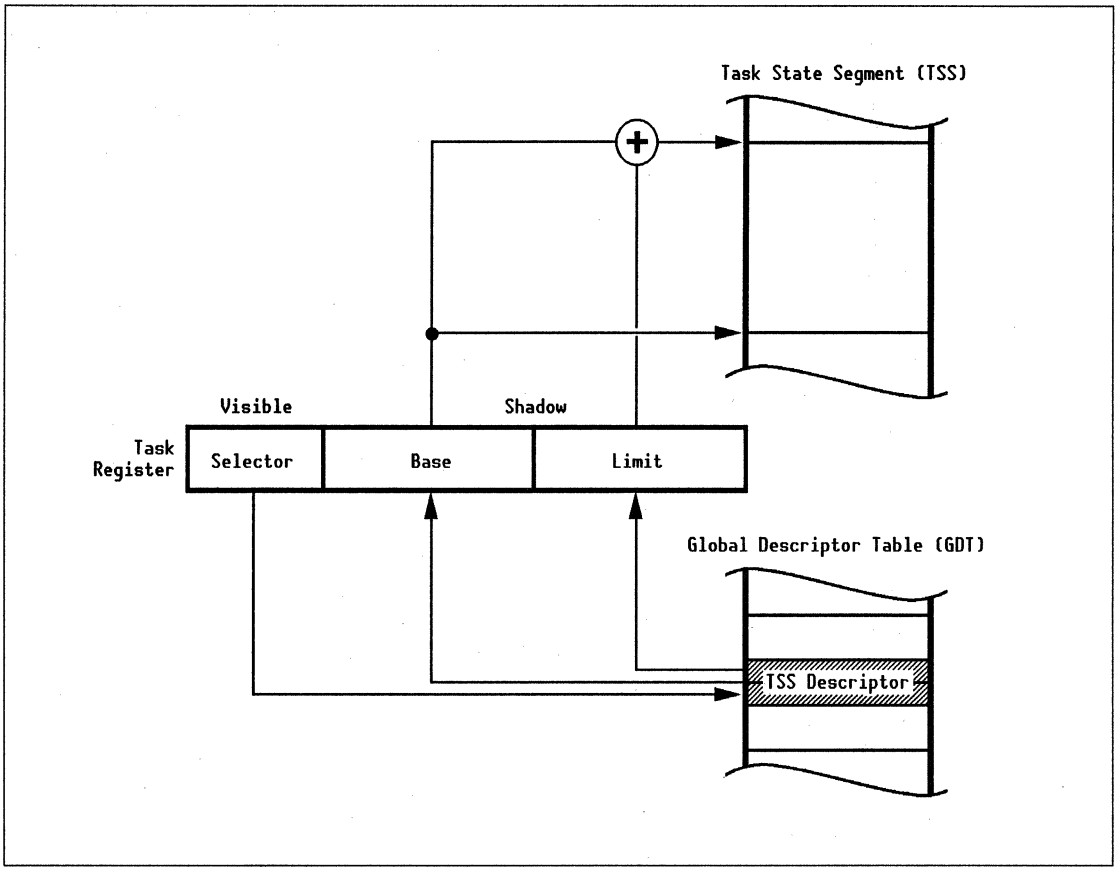
X = No change.

Task Register

The task register has a 16-bit visible part that holds the current TSS selector and a 64-bit invisible shadow register that holds the base and limit of the TSS. During a task switch, the segment selector points to a TSS descriptor in the GDT, which is then automatically loaded into the invisible (shadow) part of the task register and used to locate the TSS of the new task.

The mechanism by which the task register points to the TSS is illustrated in Figure 4-28.

Figure 4-28. TSS Selection With the TR Register



Two instructions are used to load and store the task register: LTR and STR.

LTR—The LTR instruction loads the visible part of the TR register with a register or memory operand, a selector for a TSS, which must be an index to a TSS descriptor in the GDT. The TSS descriptor addressed by the TR register is then loaded automatically from the GDT into the TR shadow register, and the busy bit (bit 9) of the TSS descriptor is set to 1.

STR—The STR instruction stores the visible part of the TR register (the segment selector, but not the corresponding descriptor in the shadow register) in a register or memory operand.

Task Gates

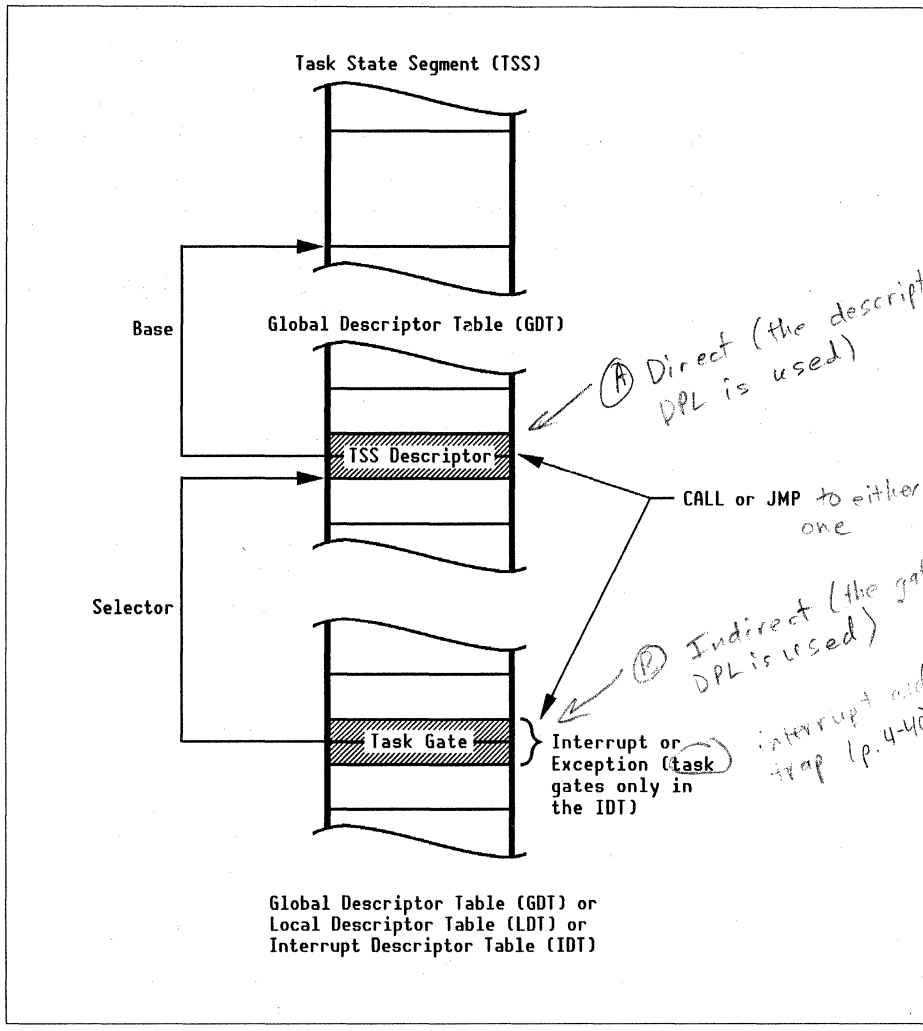
Like interrupt gates, trap gates, and call gates, task gates are descriptors that point to other descriptors. Task gates point to TSS descriptors and have their own DPL. Thus, task gates manage indirect access to task state segments on the basis of privilege level.

Task gates can be stored in the GDT, the IDT, or an LDT, as shown in Figure 4-29. Calls, jumps, interrupts, and exceptions can force task switches by accessing task state segments either directly, by referencing the TSS descriptor, or indirectly by referencing a task gate. When task gates are used to reference indirectly, the DPL of the requested TSS descriptor is not used; instead, the task gate's DPL is used. The task gate bars access to the requested TSS descriptor, except when the CPL or the gate's RPL is less than or equal to the gate's DPL:

$$\text{Max}\ (\text{CPL}, \text{RPL}_{\text{gate}}) \leq \text{DPL}_{\text{gate}}$$

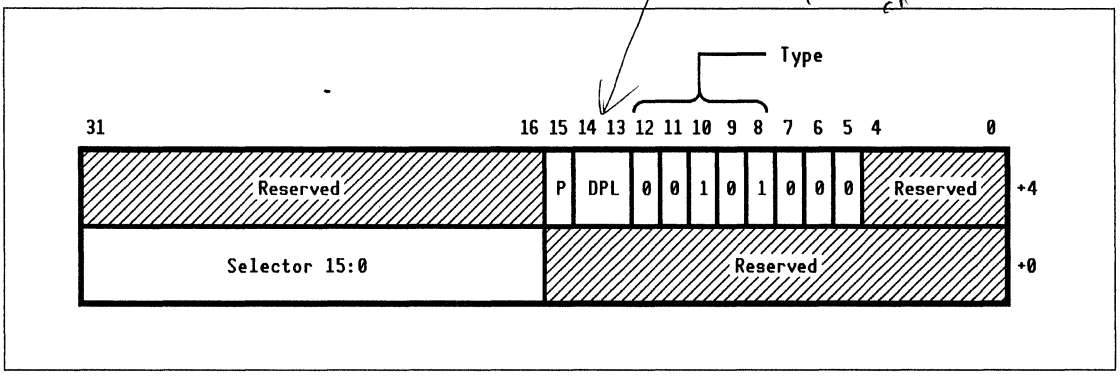
When task gates are placed in the local descriptor tables with different DPLs, they can provide access control from any task to any other task. Because they can be stored in the interrupt descriptor table, they allow interrupts and exceptions to trigger task switches. To allow a return to the interrupted task, the IRET instruction causes a task switch, if the service routine was originally called with a task switch. This will be indicated by the NT flag set to 1.

Figure 4-29. *Task Gate Mechanism*



The format of a task gate descriptor is shown in Figure 4-30. Like other control descriptors, it has only a small subset of the fields found in segment descriptors; among them are DPL and the present (P) bit. In addition, it has a segment selector that indexes to a TSS descriptor, imposing an extra level of indirection during a task switch. The RPL is never used for indexing in any segment selector. A check of the task gate's DPL is performed during a task switch through a task gate. This check replaces the check of the TSS descriptor's DPL.

Figure 4-30. Task Gate Descriptor



(+4 is high dword, +0 is low dword)

15	(+4)	P	Present—If set, this attribute indicates that the gate descriptor is valid. If this bit is clear, an attempt to access the gate causes an exception. 1 Present (valid) 0 Not present (invalid).
14:13	(+4)	DPL	Descriptor Privilege Level—These bits indicate the privilege level of the descriptor. DPL determines the minimum privilege level needed to access the segment descriptor to which the gate descriptor points. 11 Privilege level 3 (lowest) 10 Privilege level 2 01 Privilege level 1 00 Privilege level 0 (highest).
12:8	(+4)	Type	Type—These bits indicate the type of gate descriptor: 00101 Task gate
15:0	(+0)	Selector	TSS Segment Selector—This is a selector for a TSS descriptor in the GDT.

Task Switching (Dispatching)

Other than task-based interrupt and exception service routines, the processor does not automatically schedule or dispatch (switch) tasks. This is left to the operating system.

Following reset, there is no current task. System software writes an image of a TSS in memory, loads its segment descriptor (marked "not busy") into the the global descriptor table, and executes the LTR instruction to load the task register with the selector for this TSS (marked "busy"). The first task switch after the completion of system initialization will then copy the current state into the task state segment. After the operating system creates a TSS descriptor, the processor manages the "busy/not busy" status of the TSS descriptor at task switches, although software can later change the busy bit.

In any type of task switch, the processor performs the following actions:

1. Verifies privilege—Verifies that the DPL of the TSS descriptor or the task gate is greater than or equal to the selector's RPL and the processor's CPL. Hardware interrupts and exceptions do not require this check.
2. Verifies validity of TSS—Ensures that the new task's TSS descriptor, segment, and page are present, and that the TSS has a limit greater than or equal to 67h.
3. Stores current task state—Saves the current general register, segment registers, EFLAGS, and EIP registers into the current TSS.
4. Loads new TR register—Loads the selector for the new TSS into the TR register. The selector is either taken from a task gate, or it is the operand in the jump or call instruction.
5. Loads new state registers—Loads the new values for the general registers, EFLAGS, and EIP registers.
6. Loads new CR3 and LDT selector—Loads the new page directory base address (CR3), if paging is enabled, and either the selector for the local descriptor table or a null selector if no LDT is used.
7. Sets busy (B) or available (AVL) bit—Changes the type fields for both new and old TSS descriptors to "busy" or "available," depending on whether a call/interrupt or jump instruction caused the task switch. If linkage (nesting) to a suspended task is required, the NT bit of the EFLAGS register is set to 1 and the old TSS's selector is written to the new TSS's back-link field.
8. Sets task switch bit (TS) in register CR0 to 1—Sets the TS bit in register CR0 to 1 (software can use this bit to determine whether a task switch has occured).
9. Loads new segment descriptors into shadow registers—Loads each segment's descriptor into its corresponding shadow register.
10. Clears debugging break point in DR7—Clears all local breakpoint enable bits in the debug control register, DR7.

A task switch will push an error code on the stack if certain types of exceptions cause the task switch. Also, the processor does not switch the state of a coprocessor, if present; instead, the setting of the TS bit in the CR0 register can be used to coordinate the task switch with a coprocessor or other external devices. Setting the TS bit to 1 traps coprocessor instructions.

The GDTR, LDTR, IDTR, debug registers, test registers, and control registers are not saved during a task switch. If the contents of the registers are useful to system software, the software should save them. Specifically, a page-fault service routine should save the contents of CR2, the page-fault linear address, before a task switch.

See Table 4-5 in the section entitled "Protection Mechanisms" and Table 4-4 in the section entitled "Control Gates and System Calls" for summaries of the privilege-level checking that takes place.

Table 4-8 shows the order of testing conditions during a task switch and the exceptions generated. Any exceptions generated by the first three checks occur in the context of the old task; all others occur in the context of the new task.

Table 4-8. *Exception Conditions Verified During Task Switching*

Number	Condition (If false, an exception is generated)	Vector	Exception
1	TSS descriptor present	11	Segment not present
2	TSS descriptor not busy	13	General protection
3	TSS limit ≥ 67h	10	Invalid TSS
4	Registers loaded from TSS	—	
5	New LDT selector valid	10	
6	New LDT present	10	
7	Code segment selector valid	10	
8	Code segment present	11	
9	CPL = RPL_{code} (of code just loaded from TSS)	10	
10	Stack segment selector valid	10	
11	Stack segment present	12	Stack fault
12	DPL_{stack} = CPL	10	
13	RPL_{stack} = CPL	10	
14	DS, ES, FS, GS selectors valid	10	
15	DS, ES, FS, GS readable	10	
16	DS, ES, FS, GS present	11	
17	DS, ES, FS, GS segment DPL ≥ RPL_{code} (unless the code segment is conforming)	10	

Task Memory Space

Each task may have its own memory space, protected from other tasks. Segmentation or paging (or both) can provide this protection; the task switching mechanism supports both. A task switch reloads the LDTR, which points to the current local descriptor table. The LDT defines the segments that are allocated to the task, so referencing a new one is equivalent to moving to a new memory space. A task switch also loads register CR3 with a new page directory base register (PDBR), which points to the task's page directory. This also has the effect of moving to a new memory space.

Tasks can have shared memory spaces at the segment or page level. At the segment level, all tasks share the GDT; thus, any unprotected segment mapped by the GDT is shared by all tasks. It is possible to load the same segment descriptor into more than one LDT so that more than one task can have access to the same area of the linear address space. However, because each task can have its own mapping of linear to physical addresses, this by itself is not guaranteed to result in a shared memory space, unless pages are mapped one-to-one or paging has been disabled.

At the page level, any virtual page can be mapped to any physical page. Thus, shared memory can be implemented by mapping the same physical page to the linear address space of more than one task.

Nested (Linked) Tasks

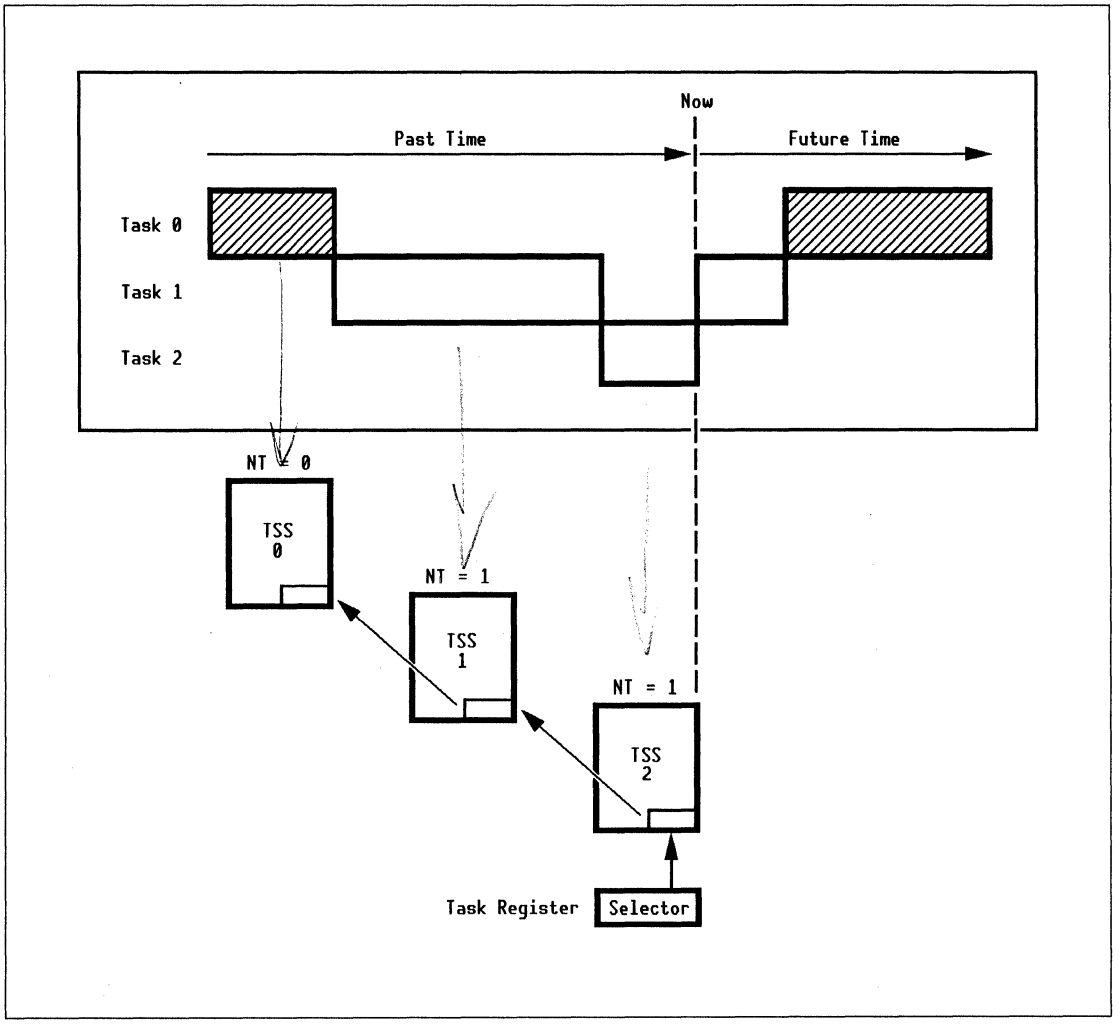
Each task can have its own LDT which can contain task descriptions to other tasks

The TSS has a back-link field that contains the segment selector of the TSS descriptor for the previous task. This allows one task to access another task via an interrupt, exception, or call. Its most common use is for interrupt and exception handling routines, so that an IRET instruction can restore the previous task state.

In non-nested task switching, the processor sets the busy bit (bit 9) of the new task's TSS descriptor and clears that bit in the old task. In nested task switching, however, the processor leaves the old task's busy bit set to 1.

The NT flag in the EFLAGS register provides the only indication of nested tasks. When set to 1, it indicates that the back-link field in the TSS for the current task contains a valid selector for the previous task. An IRET instruction will then switch to the task pointed to by the back-link field. Figure 4-31 shows task nesting and the state of the NT flag.

Figure 4-31. Nested Tasks



I/O

I/O can be implemented either in a separately addressable I/O space, using the separate set of I/O instructions, or in a memory-mapped I/O space, using the full set of general-purpose instructions.

The I/O Space

The processor's separate I/O space is a single linear-address space of 64kB, beginning at I/O address 0. Ports can be 1, 2, or 4 bytes wide. The processor provides both global and I/O-port-specific protection mechanisms for this space. Global protection is applied through two EFLAGS register bits that specify the I/O privilege level (IOPL), i.e., the maximum CPL required to execute I/O instructions. Byte-level protection is provided by the I/O permission bitmap (IOPB), a data structure in the TSS that provides access control bits for individual bytes in the I/O space. Both of these mechanisms can be used by the operating system to control calls from an I/O device. The mechanisms are described further in the sections that follow.

The chief advantage of keeping memory and I/O spaces separate is that separation offers the most reliable system protection. The execution of an I/O instruction is visible to external hardware through the M/IO* pin, and external hardware can treat the bus cycle in a special way. Reads and writes to I/O space should not be captured by a cache, because this would delay and possibly interfere with the activity of peripherals.

The only reserved addresses in the 64kB I/O space are in the range reserved for a coprocessor. When F8h and FCh are used for coprocessor accesses, the most significant bit of the address A31 is asserted. This provides external hardware a means of distinguishing a coprocessor access from an I/O access. This is because the address space between 800000F8h and 800000FFh, which is outside the I/O address space, is also used for coprocessor communication. Many system designs, however, use an I/O space smaller thn 64kB. For example, the IBM PC/AT implements a 1kB space. Specific system implementations may apply additional restrictions. For example, the PC/AT reserves all of the addresses up to FFh for standard system peripherals; user-defined peripherals must occupy the space from 100h to 3FFh. Table 4-9 shows the reserved I/O addresses on the PC/AT.

Table 4-9. *PC/AT Reserved I/O Addresses*

I/O Address	Device
00:0Fh	8237A DMA controller 1 (byte transfers, master)
20:21h	8259A interrupt controller 1 (master)
40:5Fh	8254 timer
60h and 64h	8042 keyboard logic
61h	Port B
70h	NMI mask (on writes)
70:71h	MC146818 real-time clock
80:8Fh	DMA page registers
A0:A1h	8259A interrupt controller 2 (slave)
C0:DFh	8237A DMA controller 2 (word transfers, slave)
F0h	Clear coprocessor busy
F1h	Reset coprocessor
F8:FFh	Coprocessor
100:3FFh	Expansion bus
170:177h	Hard disk 2 (WD 1010/1014/1015)
1F0:1F7h	Hard disk 1 (WD 1010/1014/1015)
200:207h	Game ports
278:27Fh	Parallel port
2E8:2EF	Serial port
2F8:2FFh	Serial port (NS16450)
300:3F1h	Prototype card
360:36Fh	Reserved
372:377h	Floppy controller 2 (NEC µPD765)
378:37Fh	Parallel port
380:38Fh	SDLC controller 2
3A0:3AFh	SDLC controller 1
3B0:3BBFh	Video (monochrome mode)
3BC:3BFh	Printer
3C0:3CFh	Video (EGA)
3D0:3DFh	Video (CGA)
3E8:3EFh	Serial Port
3F0:3F7h	Floppy controller 1 (NEC µPD765)
3F8:3FFh	Serial port (NS16450)

Memory-Mapped I/O

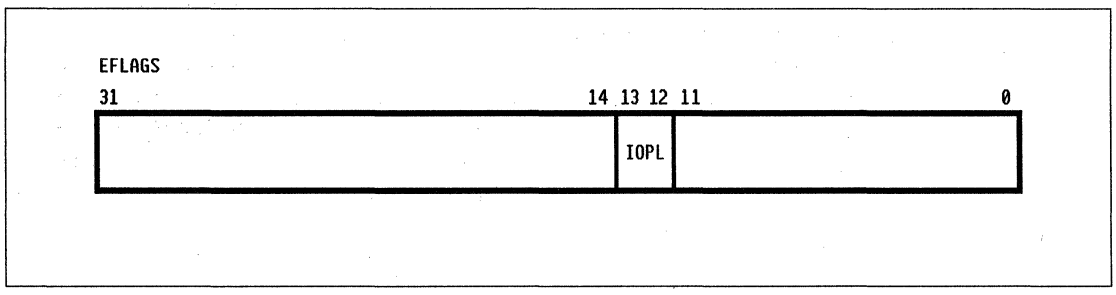
In memory-mapped I/O, external hardware can route certain memory addresses to I/O devices. From the viewpoint of software, accesses to these I/O addresses then work in the same way as ordinary memory accesses. Typically, each memory-mapped device is located in a segment of its own. Precautions must be taken, however, because memory-mapped I/O lacks the protection features provided for the I/O space.

The chief advantage of using memory-mapped I/O is that the general-purpose arithmetic and logical instructions that operate on memory-space operands can be used for accessing I/O. When a separate I/O space is used, it can be accessed only by using the special I/O instructions IN, INS, OUT, and OUTS. Memory-mapped I/O allows application software to set bits in a peripheral register without passing the contents of the peripheral register through a processor register. If memory-mapped I/O is used, it may be necessary for software to take special precautions, such as disabling a cache for regions of the memory space that are mapped to I/O peripherals.

I/O Privilege Level (IOPL)

Global protection of the I/O space is provided by the IOPL field of the EFLAGS register, shown in Figure 4-32. To execute an I/O instruction, the CPL must be less than or equal to the IOPL.

Figure 4-32. *I/O Privilege Level (IOPL)*



The following instructions must have CPL ≤ IOPL:

- IN
- OUT
- INS
- OUTS
- CLI
- STI.

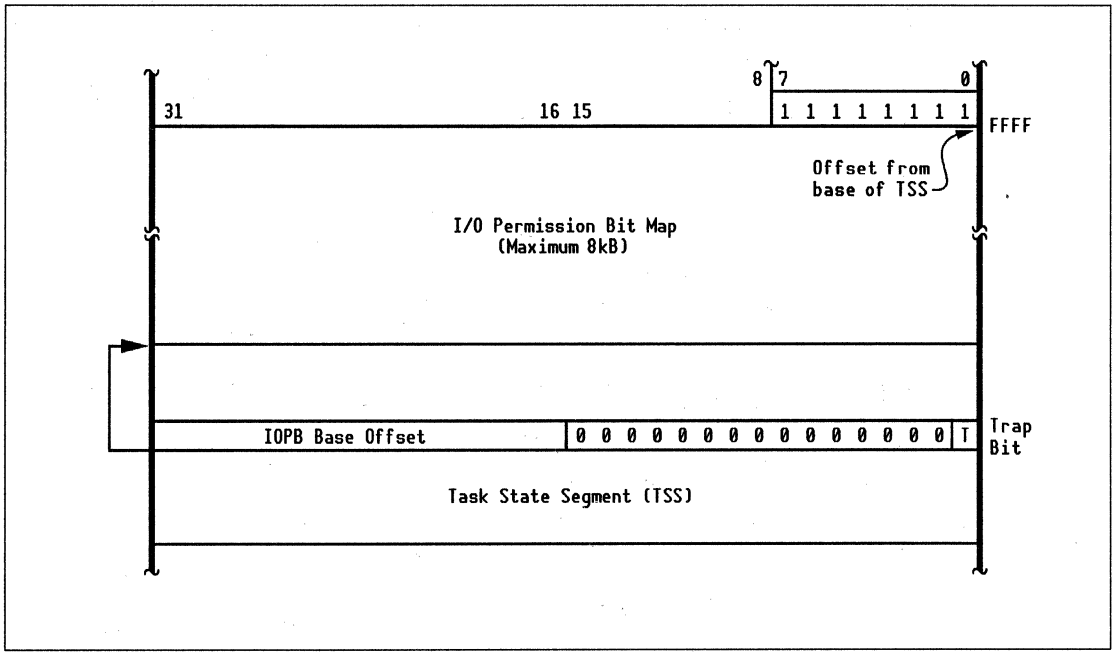
In multitasking, each task has its own copy of the EFLAGS register and can therefore have its own IOPL. The I/O protection level is not checked in real mode.

I/O Permission Bitmap (IOPB)

The I/O permission bitmap provides I/O-port-specific protection that may differ from task to task. This byte-level protection map is stored above the TSS, which contains an offset (the IOPB base offset) of the map's base from the base of the TSS. Since each task has its own TSS, access control can be mapped differently for each task. The mechanism is available only in the multitasking environment of protected mode or virtual-8086 mode.

The maximum IOPB base address is DFFFh if a full map is used. If the IOPB base address points at or past the end of the TSS, an exception is generated for any I/O operation. The map can have a permission bit for each I/O address in the 64kB I/O address space. The bits in the IOPB correspond to addresses for bytes in the I/O space, starting from 0 and covering as many addresses as needed, up to a maximum of 8kB. Access to I/O address 0 is controlled by bit 0 of the first byte of the bit map; address 1 by bit 1, and so on. If a bit in the map is cleared to 0, no protection violation is reported when the corresponding I/O address is accessed. If the bit is set to 1, any I/O reference to that address will trigger a general-protection exception. On a word or doubleword access, a bit set to 1 for any of the bytes in the operand will trigger an exception. The processor reads two bytes from the IOPB for every I/O-port access, so the minimum IOPB size is two bytes. The map must end with a byte whose bits are all 1s. Figure 4-33 illustrates the the IOPB.

Figure 4-33. I/O Permission Bit Map (IOPB)



When an I/O instruction is executed, the processor loads two bytes from the IOPB, starting with the byte that contains the permission bit for the lowest I/O address referenced by the instruction. This gives the processor access to all the permission bits that must be checked on a word or doubleword access, even if those bits straddle a byte boundary.

Is this true?

The last byte of the IOPB must have all bits that correspond to unimplemented addresses beyond the end of the I/O address space set to 1. In addition, this last byte must be included within the segment limit for the TSS. These provisions keep access to high I/O addresses from generating an exception when the processor loads the second byte from the IOPB.

Interrupts and Exceptions

Interrupts and exceptions change the normal sequence of instruction execution. They occur at the boundaries of instructions or between repeated parts of string instructions. Most of them occur transparently to the user program and force the transfer of control to another procedure or task that handles the condition and then returns control, if possible.

Interrupts are triggered primarily by hardware events, such as an I/O device request for service. In typical systems, most interrupts are signaled through the processor's interrupt request (INTR) pin by external hardware. Some unusual events, such as a power or other hardware failure, are signaled through the processor's nonmaskable interrupt (NMI) pin. In addition to these hardware events, software can also force interrupts with the INT instruction.

Exceptions are triggered exclusively by events in the execution of instructions, such as attempts to access a page that is not present in memory, or attempts to divide by zero. There are three types of exceptions, distinguished by the point in the execution stream at which the exception is reported: faults, traps, and aborts. These are discussed in the following paragraphs.

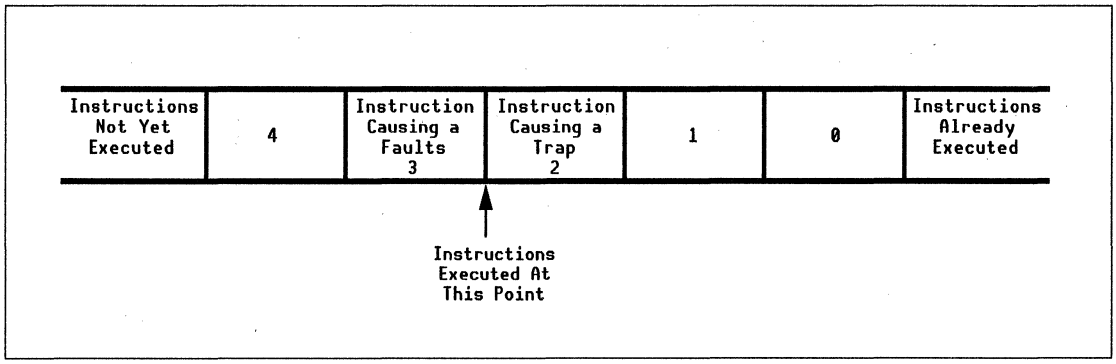
Faults—Faults restore the processor state to the faulting instruction. The faulting instruction appears not to have executed.

Traps—When a trap exception occurs, the current instruction or iteration of a string instruction completes, and the processor is left pointing to the instruction following the instruction that encountered the exception, unless the faulting trapped instruction was a string instruction. In the latter case, the processor points to the string instruction. In control-transfer instructions, the processor state is restored to the destination of the transfer, not to the next instruction located after the transferring instruction in the instruction queue.

Aborts—An abort leaves the processor at an indeterminate instruction following the faulting instruction. Aborts reflect serious errors, and the instruction cannot be restarted.

Figure 4-34 illustrates the state of instruction execution after a fault or trap.

Figure 4-34. State of Instruction Pointer After Exceptions



An instruction that causes a trap is allowed to complete before an exception is generated. An instruction that causes a fault is either not allowed to begin execution or is restored to its pre-execution state before an exception is generated. In Figure 4-34, if the exception was a trap, instruction 2 caused it. If it was a fault, instruction 3 caused it. It is not possible to know which instruction caused an abort.

Fault exceptions permit software to restore the processor state to the instruction that triggered the exception. They allow software to fix the cause of the exception and make another attempt to execute the instruction. This feature, called *instruction restart*, is necessary for implementing demand-paged virtual memory.

Demand-paged virtual memory allows parts of the memory space to be disk-resident rather than memory-resident. For example, a page that is not present in memory will have its P bit cleared to 0 in its page table entry. An attempt to read, write, or execute this page will cause a fault. The operating system then has an opportunity to allocate memory for the page, read the page from disk, update the page table entry, and return execution to the faulting instruction.

all exceptions

While there are distinctions between interrupts and exceptions, there are also many contexts in which interrupts and exceptions appear to be indistinguishable. For example, ~~the invalid opcode fault (exception 6)~~ can be invoked in software with the INT instruction, and both exceptions and interrupts are linked to their handling routines through the interrupt descriptor table.

Registers and Data Structures

Interrupts and exceptions are supported by the following memory data structures and on-chip registers:

- Vectors
- Interrupt descriptor table
- Interrupt descriptor register
- Control gates (interrupts, trap, and task).

These elements are discussed in the following paragraphs.

Vectors—A vector is a byte that identifies the cause of the event. Up to 256 vectors can be defined. The first 16 are predefined by the processor.

Interrupt Descriptor Table (IDT)—In protected mode and virtual-8086 mode, the IDT is a table containing descriptors for interrupt gates, trap gates, and task gates. Any one of these three types of gates can be accessed to branch to an interrupt or exception handler.

Interrupt Descriptor Table Register (IDTR)—The IDTR is a 64-bit register containing the base and limit of the IDT.

Control Gates—In protected mode and virtual-8086 mode, these are descriptors in the IDT that gate access to interrupt or exception handlers on the basis of privilege level.

Vectors

Each of the 256 possible interrupts and exceptions has a vector number that identifies the cause of the event. *Interrupt vectors* are generated by external hardware such as an interrupt controller. The external hardware puts the vector on the data bus, where it is read automatically by the processor during the interrupt-acknowledge cycle. *Exception vectors* are generated internally by the processor.

In protected mode, the processor uses either type of vector (multiplied by 8) as an index into the IDT to locate the appropriate handling routine. The IDT provides the link between interrupt or exception vectors and the service routines that handle the events. In protected mode and virtual-8086 mode, the processor scales the vector by 8, the number of bytes in a descriptor, to obtain the index into the IDT. In real mode, the processor scales the vector by 4 and reads a two-byte selector and offset from the IDT.

The section entitled "Summary of Interrupt and Exception Conditions" contains a complete list of all vectors that are predefined by the processor. Other vectors can be defined by the operating system.

Interrupt Descriptor Table and Register

The interrupt descriptor table (IDT) contains up to 256 eight-byte descriptors for up to three types of gates—interrupt gates, trap gates, and task gates. All descriptors are optional, although systems are rarely designed without interrupt gate descriptors. The IDT may be placed anywhere in memory. The bottom 16 descriptors, the 16 predefined interrupts and exceptions, are normally always present in physical memory; page faults could not be handled otherwise. The table is located by the 32-bit linear base address and 16-bit limit contained in the interrupt descriptor table register (IDTR). This register is loaded and stored with the LIDT and SIDT instructions, each of which has a 6-byte operand for the base and limit.

The IDT is structured like the global descriptor table, which also contains descriptors, except that all entries in the IDT contain gates; the first entry is not reserved, as in the GDT. If the vector addressing the IDT exceeds the table's limit, a second attempt to execute the faulting instruction will be made. If a double fault occurs, the processor will go into its shutdown mode and generate a special bus cycle.

Gates

Interrupt, trap, and task gates, are eight-byte descriptors. Their structure is similar to that of segment descriptors, but they themselves contain a segment selector (and in two cases, an offset) rather than the base and limit found in segment selectors. Instead of pointing directly to a segment, a gate points to another descriptor that points to a segment. By doing so, gates provide indirect access based on privilege level. The three types of gates are discussed in the following paragraphs.

Interrupt Gates—Interrupt gates contain both a selector and an offset for the handling procedure. When an interrupt gate is accessed, the processor disables instruction tracing by clearing the trap flag (TF) to 0 after pushing the current EFLAGS register on the stack. For interrupt gates, the processor also disables further maskable interrupts by clearing the interrupt flag (IF) to 0. This may be required for certain types of events, such as page faults. All flag values are restored during an IRET instruction.

Trap Gates—Trap gates are identical to interrupt gates, except that the IF flag is not changed.

Task Gates—These gates contain only a selector (not an offset) that points to a TSS, not directly to a handling procedure. No flags are changed by the processor when a task gate is used. The section entitled "Multitasking" describes task gates in more detail.

The operating system controls access by setting the DPL of each gate. When gates are used, the DPL of the requested descriptor is not used; instead, the gate's DPL is used. The gate then bars access to the requested descriptor, except when the CPL of the requestor (or in the case of a task gate, the gate's RPL) is less than or equal to the gate's DPL.

The gates are compared in Figure 4-35. For a detailed explanation of their bits, see the section entitled "Registers and Descriptors." For a details on privilege-level checking rules, see the sections entitled "Protection Mechanisms" and "Control Gates and System Calls."

Figure 4-35. *Three Types of Gates Used for Interrupts and Exceptions*

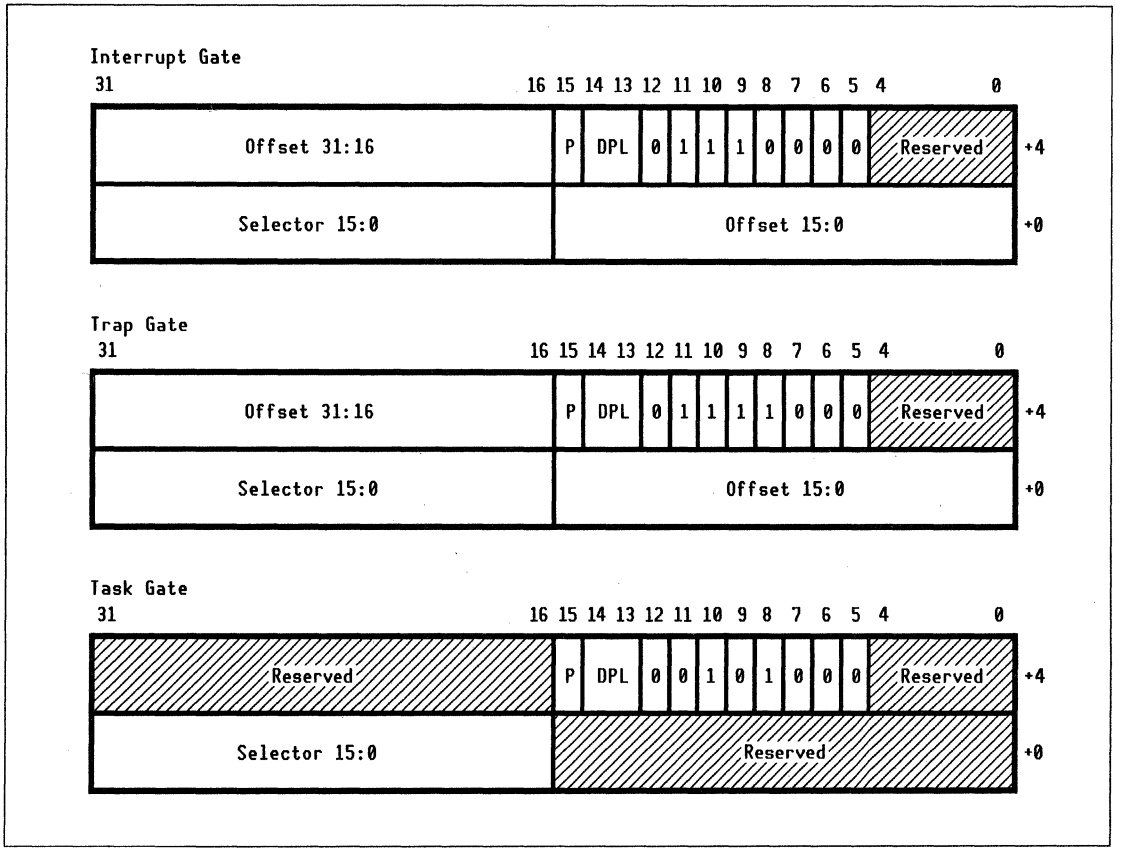


Figure 4-36 shows the mechanism for interrupt gates and trap gates, which involves gate descriptors in the IDT and code segment descriptors in the GDT. The processor scales the vector by 8, the number of bytes in a descriptor, to obtain the index into the IDT.

Figure 4-36. *Vectoring for Interrupt Gates and Trap Gates*

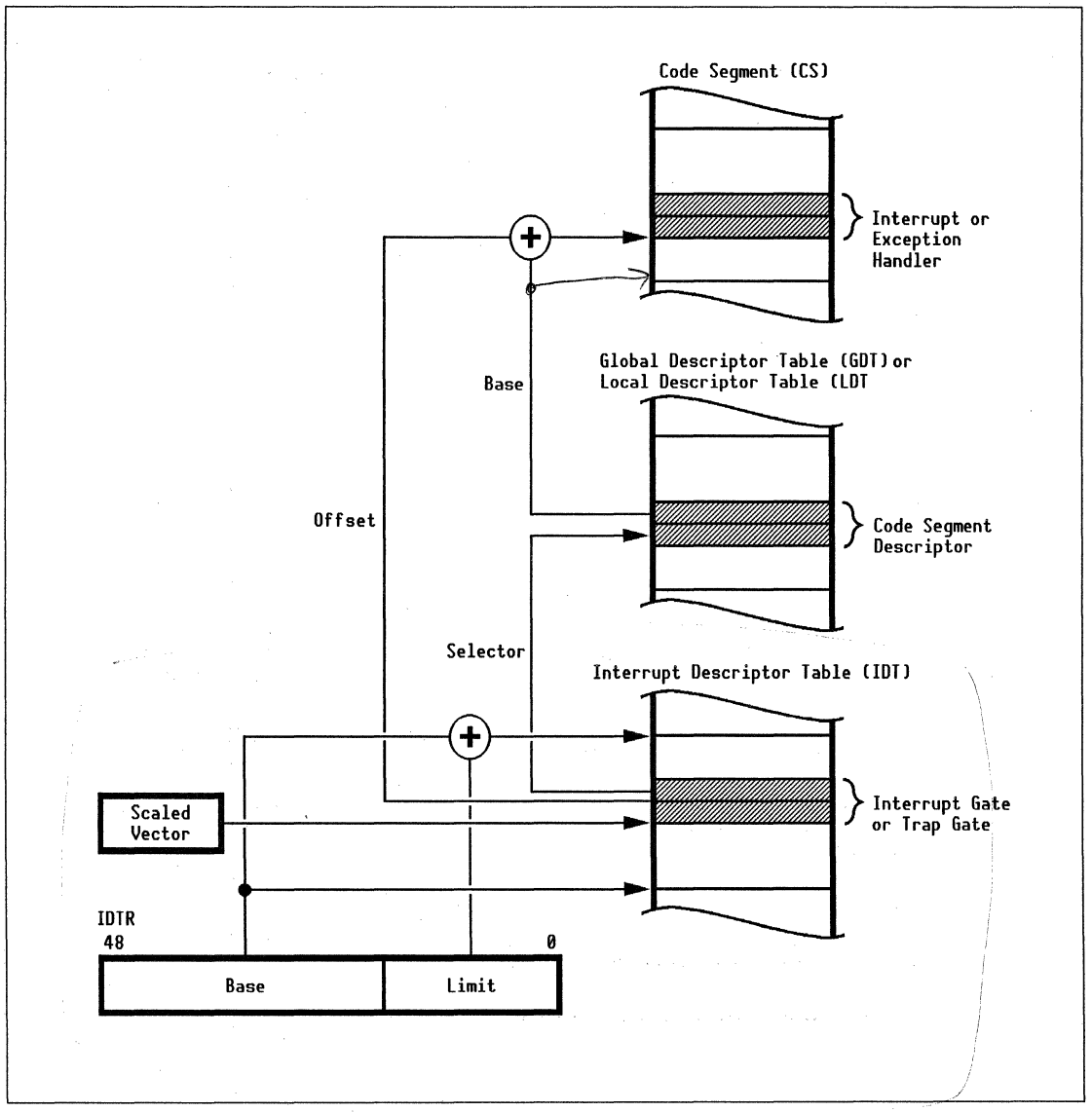
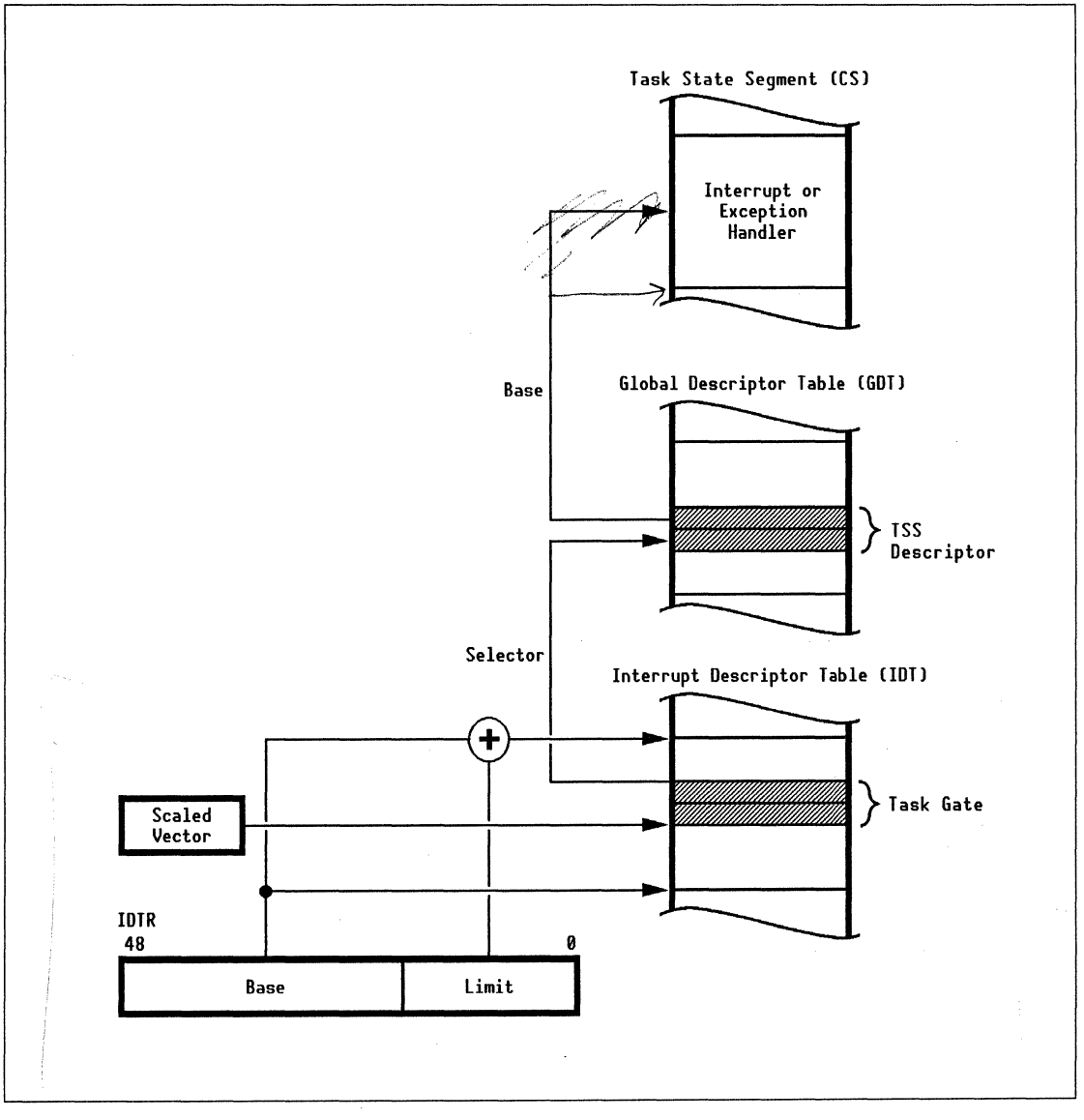


Figure 4-37 shows the analogous mechanism for task gates. Unlike the mechanism for interrupt gates and trap gates, task gates point to a TSS descriptor rather than a code-segment descriptor in the GDT. Also, task gates do not contain an offset; only a base address is needed to access a TSS.

Figure 4-37. *Vectoring for Task Gates*



Error Codes

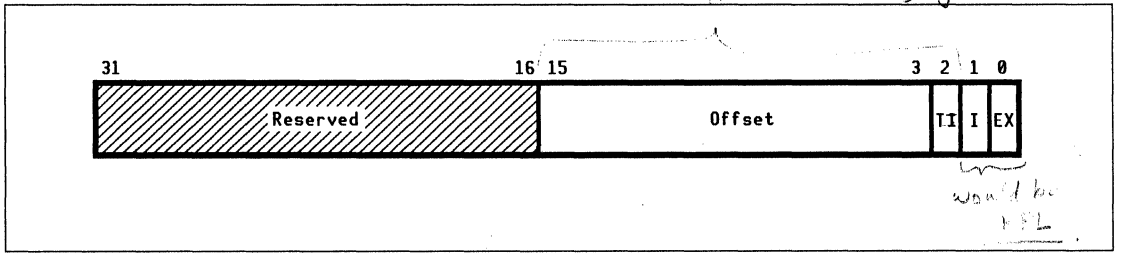
During exceptions that relate to a specific segment or to a page fault, the processor provides additional information about the event through an error code. Error codes from 32-bit (Super386) gates are pushed onto the stack of the exception handler as doublewords, to conform with the 32-bit stack pushes of the code segment and EIP. Error codes from 16-bit (80286) gates are pushed onto the stack of the exception handler as words. Table 4-10 lists the exceptions that provide error codes.

Table 4-10. *Events With Error Codes*

Vector	Description	Type
8	Double fault (error code = 0)	fault abort
10	Invalid task state segment	fault
11	Segment not present	fault
12	Stack fault	fault
13	General protection	fault / trap
14	Page fault (special format)	fault
17	Alignment Check (error code = 0)	fault.

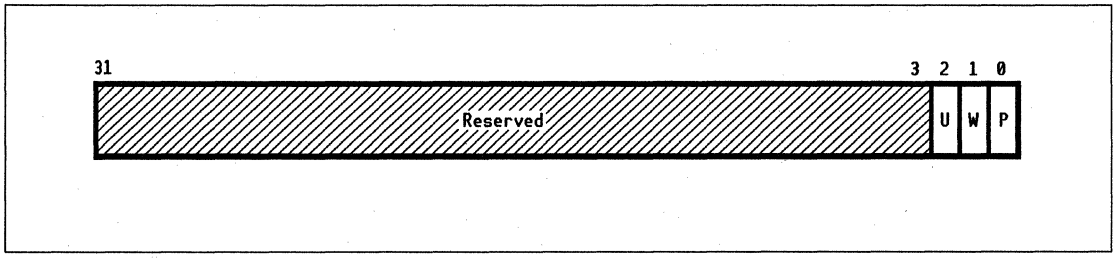
The error codes have one of two formats. Figure 4-38 shows the format for exceptions (10, 11, 12, and 13) that relate to a specific segment. Figure 4-39 shows the format for a page-fault exception (14).

Figure 4-38. *Error Code Formats (Except Page Faults)*



bits:	15:3	Offset	Offset—These offset bits indicate the descriptor-table index for the segment from which the event arose.
	2	TI	Table Indicator—This bit indicates the descriptor table in which the descriptor is located: 1 LDT 0 GDT
	1	I	IDT Override—This bit overrides the TI bit to indicate the descriptor table in which the descriptor is located: 1 IDT 0 TI bit indicates the descriptor table.
	0	EX	Exception Error—This bit indicates a secondary exception generated by an atempt to access the IDT in order to invoke another exception. 1 Secondary exception 0 All other cases.

Figure 4-39. *Error Code Format (Page Faults)*



bits:	2	U	User/Supervisor—This bit indicates the current privilege level (CPL) when the event occurred. 1 User mode (privilege level 3) 0 Supervisor mode (privilege level 0, 1 or 2).
	1	W	Write/Read—Bit W indicates whether the event was generated by a write or a read. 1 Write 0 Read.
	0	P	Present/Page-Protection—This bit indicates whether the event was caused by a not-present page table or page directory entry, or by a page-level protection violation. 1 Page-protection violation 0 Not-present table or directory entry.

Interrupt and Exception Handlers

As illustrated in Figures 4-38 and 4-39, an interrupt or exception servicing routine (or *handler*) can be implemented as either a procedure or a task. These two approaches can be characterized as follows:

Procedures—Accessing a handler through an interrupt gate or trap gate causes the handler to be run as a procedure in the same context as the current task. Interrupt and trap gates are dispatched by the processor and go directly to the handling procedure. Exception handlers are typically implemented as procedures so that the handling can be done in the context that generated the exception (although this has the potential disadvantage of not guaranteeing a clean context). By avoiding the overhead of a task switch, handling latency is minimized.

Tasks—Accessing a handler through a task gate causes the handler to be run as a new task, in a new context with its own stack. Tasks are dispatched by the processor. Registers are saved and restored automatically, without operating system intervention.

Interrupt handlers often have the most to gain from implementation as tasks, because most interrupts (such as I/O-device requests for service) do not need the data available in the old context. Interrupt, exception, and IRET instructions can switch tasks at any privilege level. Interrupt tasks have their own context, so they can issue operating system calls and create resources freely, if resources are managed on a task-by-task basis. Procedures, on the other hand, are dependent on the resources of the tasks in which they are running. In spite of these advantages, the overhead of a task switch imposes a latency that may not be acceptable for critical real-time environments.

The following sections explain how each approach works.

Procedure-Based Implementation

When implemented as procedures, interrupt and exception handlers are treated by the processor in much the same way as calls through a call gate. A privilege-level comparison is made between the handler's code segment (pointed to by the interrupt or trap gate) and the current privilege level of the interrupted task or procedure. If the handler's code segment is more privileged than the accessor, a new stack is created; otherwise, the handler uses the stack of the interrupted procedure.

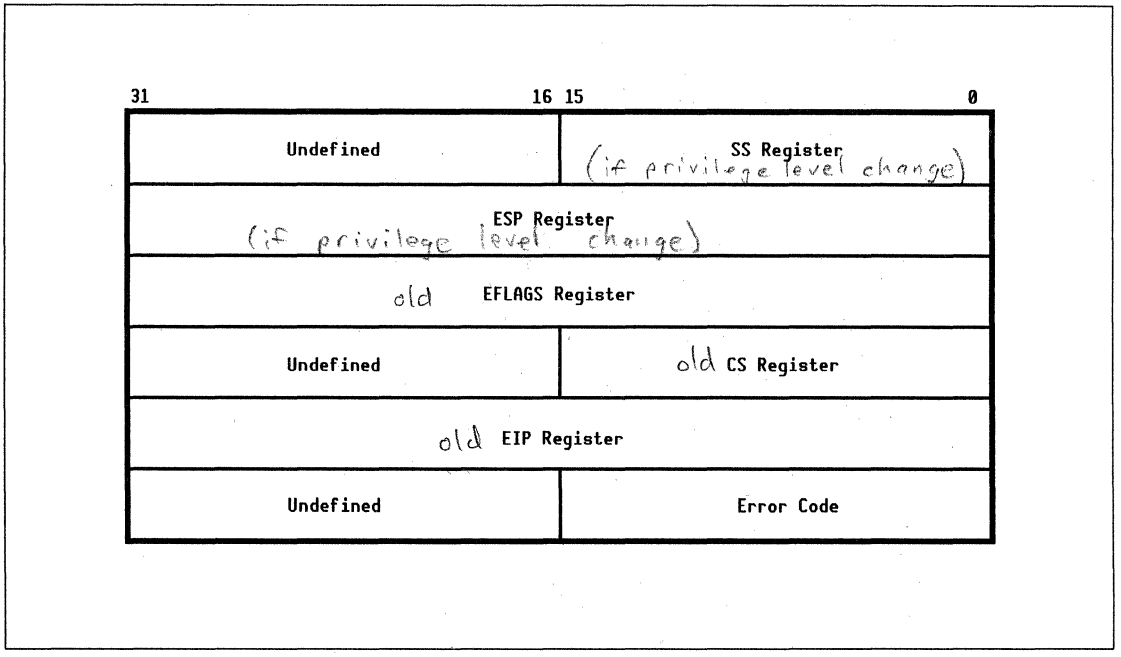
If a stack switch occurs

The processor pushes several things on this stack, in the following order:

1. Old stack segment (SS) register, if there is a privilege-level change
2. Old stack pointer (ESP) register, if there is a privilege-level change
3. Old EFLAGS register
4. Old code segment (CS) register
5. Old instruction pointer (EIP) register
6. Error code for the event, if it is generated.

Figure 4-40 shows the top of the handler's stack immediately following entry into the handling routine when the privilege level changes and an error code is provided (i.e., maximum number of things from the above list pushed onto the stack). The stack grows down. To keep the stack aligned to doubleword addresses, the 16-bit SS and CS registers are pushed onto the stack as the lower half of doublewords, with the upper words undefined.

Figure 4-40. *Stack Frame for Interrupt or Exception Procedure*



The handler returns by executing a 32-bit IRET instruction. Because IRET expects to see the saved EIP on the top of the stack, the routine must pop the error code or adjust the stack pointer before returning, if such an error code is on the stack. If the routine involved no privilege-level change, the processor then pops the EIP, CS, and EFLAGS values into their registers. If a privilege-level change occurred, the processor also pops the ESP and SS values, thereby transitioning to the old stack. At this point, the processor executes the next instruction of the old procedure, which will be determined by the type of event that occurred. Fault exceptions restart the instruction that caused the fault; trap exceptions and interrupts execute the next instruction after the one causing the trap or interrupt.

Task-Based Implementation

The processor automatically dispatches a new task (that of the handler) when it vectors to a task gate. The processor's sequence is:

1. Save Old TSS—Save the suspended task's context in its TSS.
2. Load New TSS—Load the handler's TSS. The saved EFLAGS values of the handler's TSS determine whether further interrupts are enabled or disabled.
3. Set Nested Task Flag—Set the nested task (NT) flag to 1.
4. Fill Back-Link Field—Fill the back-link field of the handler's TSS with the selector for the suspended task's TSS.
5. Transfer—Transfer control to the handler.

To return to the suspended task, the handler pops any error codes that were pushed onto the stack and issues a 32-bit IRET instruction. When ~~this~~ the IRET occurs, the processor's sequence is:

1. Clear Nested Task Flag—Copy the NT flag to an internal register and clear the flag to 0.
2. Save Old TSS—Save the handler's context in its TSS.
3. Load New TSS—Using the selector in the back-link field of the handler's TSS, find and load the suspended task's TSS.
4. Transfer—Transfer control to the suspended task.

As with procedure-based handlers, the processor then executes the next instruction of the old task. For fault exceptions, it restarts the instruction that caused the fault. For trap exceptions and interrupts, it executes the next instruction after the one causing the trap or interrupt.

Task-based handlers must be implemented with care to avoid conflicts between the processor's dispatching and the operating system's task dispatching. In particular, the operating system may need to consider situations in which the handler makes a system call that causes another task switch. Prior to doing so, the operating system must be informed that a new task is running.

How?

Summary of Interrupt and Exception Conditions

Table 4-11 shows the vectors, types, level, and causes of all interrupts and exceptions defined by the processor. Other vectors can be defined by the operating system. Table 4-12 shows the same information for the IBM® PC/AT architecture.

The "Type" column in both tables indicates one of the following:

- Interrupt—An interrupt caused by external hardware
- Fault—An exception that is a fault
- Trap—An exception that is a trap
- Abort—An exception that aborts execution.

The "Error Code" column in Table 4-11 indicates whether or not an error code is pushed onto the stack of the interrupt or exception service routine. Error codes provide more specific information about the cause of the event. The structure of the error code is described in the section entitled "Error Codes."

Table 4-11. *Super386 Interrupts and Exceptions*

Vector	Description	Type	Error Code	Cause
0	Division by zero	Fault	No	Occurs when a DIV or IDIV divisor is 0, or the quotient is too long to fit into the result operand.
1	Debug exception	Fault/trap	No	This is a fault when triggered by an instruction breakpoint or a general-detect condition. It is a trap when triggered by a data address breakpoint, single-step trap, or a task switch with a T bit set to 1 in the task state segment. The DR6 register indicates the fault or trap condition. More than one exception condition may be indicated, with several bits set to 1 in DR6. Debug faults (not traps) are disabled for one instruction if the RF flag in the EFLAGS register is set to 1.
2	NMI interrupt	Interrupt	No	Occurs when external hardware asserts the nonmaskable interrupt signal and an NMI handler is not already executing.
3	Breakpoint (INT 3)	Trap	No	Occurs when an INT 3 instruction is encountered. This instruction is a one-byte form of the INT *n* instruction that can be inserted into programs as a breakpoint trap.
4	Overflow (INTO instruction)	Trap	No	Occurs when an INT0 instruction is executed when the overflow flag (OF) is set to 1.
5	Bound range exceeded	Fault	No	Occurs when a BOUND instruction determines that an array index is outside the specified array bounds.
6	Invalid opcode	Fault	No	Occurs when a bit pattern is not recognized as an instruction. This could be an invalid opcode, a register operand where a memory operand is required, or a LOCK prefix before an instruction that cannot be locked.
7	Coprocessor not available	Fault	No	Occurs on a WAIT or ESCAPE instruction when both the TS and MP bits in the CR0 register are set to 1. It also occurs if a WAIT or ESCAPE instruction is executed when the EM bit in the CR0 register is set to 1.
8	Double fault	Abort	Yes	Occurs when an exception is reported while another exception is being processed. The error code is 0. In real mode, a double fault always leads to a shutdown. In protected mode, the processor will try executing yet another instruction after a double fault before shutting down. Double faults can be handled without a shutdown by switching tasks or otherwise getting a new stack.
9	Coprocessor segment overrun not in 486 or Pentium,	Abort	No	Indicates that a floating-point operand is triggering an unrecoverable segment limit violation. It occurs when the operand runs off the end of the address space and wraps around from the top of the address space to the bottom. The coprocessor must be reinitialized with the FINIT instruction before returning from the exception service routine. The CS and EIP registers will point to the aborted instruction.

Table 4-11. Super386 Interrupts and Exceptions (continued)

Vector	Description	Type	Error Code	Cause
10	Invalid task state segment	Fault	Yes	Occurs when a task switch to an invalid TSS is attempted. The error code contains the segment selector of the invalid TSS.
11	Segment not present	Fault	Yes	Occurs when loading a segment selector for a segment descriptor with a present bit (P bit) cleared to 0. The error code contains the segment selector for the descriptor, with the P bit cleared to 0. This fault can also be triggered when an LDT segment selector is loaded with the LLDT instruction, or when any descriptor (other than a stack descriptor) is used with the P bit cleared to 0.
12	Stack fault	Fault	Yes	Occurs when there is a limit violation during a stack segment reference (error code is 0) or during a call or interrupt to a more privileged level (error code is a selector for the stack at that level), or when loading into the stack segment register a selector that references a descriptor with a P bit cleared to 0 (error code contains the faulting segment selector). The exception service routine can determine the cause by examining the segment descriptor in question.
13	General protection	Fault	Yes	Occurs under miscellaneous circumstances, not covered by other categories, when an application program executes a privileged instruction or I/O reference. The major circumstances are listed in the section "Conditions Causing General Protection" following this table. The error code depends on the condition. If the fault was triggered by loading a segment register, the error code contains the faulting segment selector. All other conditions result in an error code of 0.
14	Page fault	Fault	Yes	Occurs during address translation when the page directory or page table entry has its present bit (P bit) cleared to 0 or when the access is not allowed by the page attributes (e.g., an attempt to write on a read-only page). The faulting linear address is placed in the CR2 register. The error code for a page fault indicates (a) whether the exception was due to a not-present page or an access rights violation, (b) the privilege level of the task, and (c) whether the access was a read or write.
15	Reserved			
16	Coprocessor error	Fault	Yes	Occurs during a coprocessor or WAIT instruction when the result of a floating-point operation causes the ERROR* signal to be asserted. The fault can only be raised if the EM bit in CR0 is cleared to 0 (no emulation) and is not reported until the next coprocessor or WAIT instruction (after the instruction that generated the error) is executed.
0-255	Interrupt instructions	Trap	No	Generated by an INT *n* instruction (opcode CDh), which can also be used to raise the predefined interrupts, 1 through 16.
	Hardware maskable interrupts	Interrupt	No	Generated by an active INTR pin. The vector is supplied by the external interrupt controller.

Condtions Causing General-Protection Faults

Some conditions that cause general protection faults are the following:

- Violating the rules of privilege
- Loading a data segment register with a system segment selector
- Loading a data segment register with a code segment selector
- Loading the stack segment register with a read-only segment selector
- Memory access with a null selector loaded in the segment selector
- Reading an execute-only code segment
- Writing a read-only segment
- Transferring control to a non-code segment
- Accessing beyond a segment limit
- Accessing beyond a descriptor table limit
- Enabling paging in CR0 when protection mode is disabled
- Issuing an instruction longer than 15 bytes
- Task switch to a busy task
- Interrupt/exception via a trap or interrupt gate to a service routine with a DPL > 0 in virtual-8086 mode.

Error codes in general-protection faults contain a selector that may be taken from the operand of the faulting instruction, the gate referenced by the instruction, or a TSS.

IBM PC/AT Interrupt and Exception Vectors

The IBM PC/AT uses a different set of vectors, which (in some cases) conflict with Super386 processor exception vectors. In PC/AT-compatible systems, software that enables protected mode must first reprogram the interrupt controllers to relocate the peripheral interrupt vectors to other addresses. Table 4-12 shows the standard PC/AT interrupt and exception vectors.

Table 4-12. *PC/AT Interrupt and Exception Vectors*

Vector	Description	Type
0	Division by zero	Fault
1	Single-step exception	Fault/trap
2	NMI interrupt	Interrupt
3	Breakpoint (INT 3)	Trap
4	Overflow (INTO instruction)	Trap
5*	Print screen	System call
6	Invalid opcode	Fault
7	Coprocessor not available	Fault
8*	8254 timer	Interrupt
9*	8042 keyboard	Interrupt
0Ah	Video vertical retrace routine	Interrupt
0Bh	IRQ3 (expansion bus, serial port 1)	Interrupt
0Ch	IRQ4 (expansion bus, serial port 2)	Interrupt
0Dh	IRQ5 (expansion bus, parallel port 1)	Interrupt
0Eh	IRQ6 (expansion bus, floppy disk)	Interrupt
0Fh	IRQ7 (expansion bus, parallel port 2)	Interrupt
10-1Fh	BIOS services	System call
20-27h	DOS services	System call
28-3Fh	Reserved for DOS	—
40-5Fh	Reserved	—
60-67h	Available for user programs	—
68-6Fh	Not used	—
70h	6818 real-time clock	Interrupt
71h	IRQ9 (expansion bus, video retrace)	Interrupt
72h	IRQ10 (expansion bus)	Interrupt
73h	IRQ11 (expansion bus)	Interrupt
74h	IRQ12 (expansion bus)	Interrupt
75h	Coprocessor error interrupt	Interrupt
76h	IRQ14 (expansion bus, hard disk)	Interrupt
77h	IRQ15 (expansion bus)	Interrupt
78-7Fh	Not used	Interrupt
80-85h	Reserved for BASIC interpreter	—
86-F0h	Used by BASIC Interpreter	—
F1-FFh	Not used	System call

* Conflicts with the standard Super386 vectors in Table 4-11.

The PC/AT architecture uses interrupt vectors to access the BIOS in the system ROM, operating system, and BASIC interpreter services. These calls are made by using the INT *n* instruction. For compatibility with the PC/XT, the PC/AT does not use the standard coprocessor error exceptions (9h and 10h). Instead, it reports errors though an interrupt request line (75h).

Simultaneous Interrupts or Exceptions

When the processor detects an interrupt or exception, it attempts to store the state of the processor and jump to the interrupt or exception handler. It is possible that the processor will encounter another interrupt or exception while attempting to do these operations. When this happens, the interrupts or exceptions are checked and reported in a priority sequence. The highest priority event is checked and reported, and other events are either deferred (interrupts) or lost (exceptions). If any exception condition is still true after the service routine of a higher priority event executes, it may be reported on the next attempt to execute the faulting instruction.

Some exceptions are not possible while vectoring to a handler. For example, the processor does not perform a divide operation, and hence cannot encounter a divide exception. It is possible for the processor to encounter a stack fault, not-present, or general-protection exception. These exceptions are considered contributory, and if encountered during an attempt to process a contributory exception, they will generate a double fault exception. Any exception encountered while the processor is attempting to invoke the page fault handler (exception 14) will generate a double fault.

Table 4-13 shows the sequence in which interrupts and exceptions are checked and reported.

Table 4-13. *Interrupt and Exception Priority*

Priority	Interrrupt/Exception	Description
1	Debug traps	Checks the instruction that has just completed. Include cases as follows: a. The trap flag is set to cause a single-step. b. An operand of the previous instruction had a debug match. c. The T bit in the TSS was set for the task switch just completed.
2	Debug faults	Checks the instruction that is about to execute. Generates exception 1.
3	ANMI interrupt (SuperState V)	Checks the alternate nonmaskable interrupt input signal. This is one of the entry mechanisms to the Super386 processor's SuperState V mode.
4	NMI interrupt	Checks the nonmaskable interrupt input signal.
5	INTR interrupt	Checks the maskable interrupt request input signal.
6	Segmentation fault	Checks for faults that prevent the next instruction from being fetched. This check generates exceptions 11 and 13.
7	Translation fault	Checks for faults that prevent the next instruction from being fetched. This check generates exception 14.
8	Decoding fault	Checks for faults encountered while decoding the next instruction. Faults include: a. Invalid opcode (exception 6 for opcodes that do not exist or are not valid in the current execution mode); b. Instructions that are longer than 15 bytes; or c. Instructions that are not valid at the current privilege level (exception 13).
9	Coprocessor WAIT or ESCAPE	Checks for the following conditions, in this order: a. If WAIT instruction, generates exception 7 if TS = 1 and MP = 1 in the CR0 register. b. If WAIT or ESCAPE instruction (D8-DF), generates exception 7 if TS = 1 and MP = 1 in the CR0 register. c. If WAIT or ESCAPE instruction (D8-DF), generates exception 16 if ERROR signal from coprocessor is active.
9	INT 3 or *n*	Checks for INT 3 or *n* instruction and generates interrupt 3 or *n*.
9	INTO	Checks for INTO instruction and generates interrupt 4 if the overflow flag is set.
10	Memory operand	Checks all portions of memory operands, including multiple operands, for the following conditions, in the following order. If there are multiple operands, steps a and b are performed on the first operand before any steps are performed on subsequent operands: a. Segmentation faults. Generates exception 11, 12, or 13. b. Paging faults. Generates exception 14. If a segmentation or translation fault is detected for only part of an operand, no bus access is generated. Thus, an operand only partially existing on a non-present page will not fetch or store the part of the operand that is on the present page.

Table 4-13. Interrupt and Exception Priority (continued)

Priority	Interrrupt/Exception	Description
11	DIV, IDIV, and AAM	Checks for a DIV, IDIV, or AAM instruction. Generates exception 0 if a divide by zero is attempted, or if the result cannot be represented in the destination operand size.
11	BOUND	Checks for a BOUND instruction, and generates exception 5 if the register operand exceeds the bound indicated by the two memory operands.
11	Segment selectors in control transfer	If the operation is a control transfer, checks for segment selectors that exceed table limits, are null, point to invalid or inappropriate descriptors, are not present (including gates not present), violate privilege rules, or have an instruction pointer that exceeds the segment's limit. See Table 4-8, "Exception Conditions Verified During Task Switching."

Disabling Interrupts

Several conditions temporarily block the handling of some or all interrupts. These conditions are the following:

IF Flag—The interrupt enable flag in the EFLAGS register (IF) disables maskable interrupts when cleared to 0. Exceptions and NMI interrupts are not affected.

RF Flag—The resume flag in the EFLAGS register (RF) disables debug faults for one instruction when set to 1. The RF flag is automatically cleared to 0 after one instruction is executed.

NMI Interrupt—The NMI interrupt is disabled while the NMI service routine is executing so that an NMI can never interrupt the processing of a previous NMI. After execution of an IRET instruction, NMI interrupts are automatically re-enabled.

MOV or POP Instruction—A move or pop instruction that loads the stack segment (SS) register inhibits all interrupts and exceptions until the end of the following instruction. This allows a new stack segment and stack pointer to be loaded without the risk of an interrupt between the two loads. If an interrupt were allowed to occur, the instruction pointer could be pushed into a space addressed with the new stack segment and the old stack pointer.

Interrupt-Related Instructions

Several instructions are provided for calling, managing, and returning from interrupts:

- CLI—Clears the interrupt enable flag (IF) to 0 in the EFLAGS register, disabling maskable interrupts.
- STI—Sets the IF flag, enabling maskable interrupts.
- INT *n*—Triggers an exception with a vector specified by an 8-bit immediate operand.
- INT 3—Triggers an exception with vector 3h.
- INTO—Triggers an exception with vector 4h, if the overflow flag (OF) is set to 1.
- BOUND—Triggers an exception with vector 5h, depending on the state of three operands. One operand is an array index, and the other operands are the upper and lower array bounds. If the index is outside of the bounds, the exception is called.
- IRET—Terminates the service routine and passes control back to the procedure or task that was interrupted. If the nested task flag (NT flag) is set to 1, a task switch occurs. If NMIs are disabled, they are re-enabled.
- LIDT—Loads the IDTR register from memory. This instruction is used to initialize the register with a pointer to the IDT.
- SIDT—Writes the IDTR register to memory.
- POP SS and MOV SS—Inhibit interrupts until after the following instruction completes execution. This prevents use of an invalid stack.

See p A-100

Initialization

Initialization is a procedure that causes program execution to begin in a predictable manner. The processor begins to execute in 8086-compatible real mode. System code then establishes in memory the base registers and control tables needed to support full operations.

Reset

Initialization begins with hardware external to the processor activating the RESET* signal. Hardware typically holds the RESET* signal active while

- Power is stabilizing.
- System software or the user is forcing a reset.

After reset, registers are set to the default values shown in Table 4-14. Both memory protection (segmentation) and paging are disabled. Table 4-14 shows the states of registers that have defined values after reset. The states of all other registers are undefined.

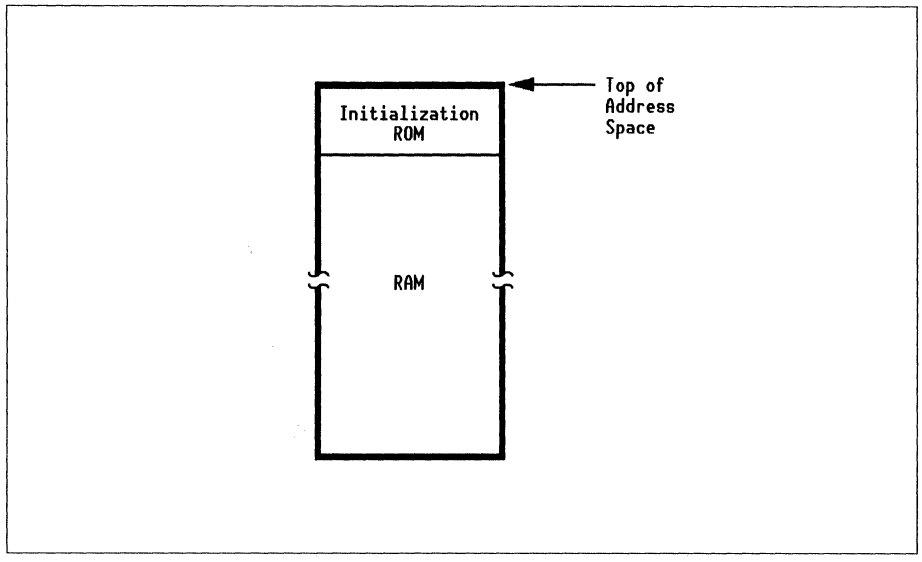
Table 4-14. State of Registers After Reset

Register	Value	Shadow Register		Function
		Base	Limit	
EIP	0000FFF0	—	—	
EFLAGS	XXXX0002	—	—	Interrupts and single-stepping disabled.
EAX	????????	—	—	Clear = passed; nonzero = failure signature.
EDX	XXXX03??	—	—	80386-compatible; revision level.
CS	F000	FFFF0000	FFFF	Addresses top 64kB of memory.
DS	0000	00000000	FFFF	Addresses bottom 64kB of memory.
SS	0000	00000000	FFFF	Addresses bottom 64kB of memory.
ES	0000	00000000	FFFF	Addresses bottom 64kB of memory.
FS	0000	00000000	FFFF	Addresses bottom 64kB of memory.
GS	0000	00000000	FFFF	Addresses bottom 64kB of memory.
IDTR	—	00000000	03FF	Compatible with 8086.
DR7	00000000	—	—	Breakpoints disabled.
CR0	????????[1]	—	—	Protection and paging disabled.

X Undefined; all registers not listed are also undefined.
? Defined, but variable among Super386 processor types.
1 FFFFFFF2 if a math coprocessor is present; FFFFFE0 if there is no math coprocessor.

The default values in the EIP register, the code segment (CS) register, and the data segment (DS) register, together with the segment descriptors in the segment shadow registers, cause code execution to begin 16 bytes below the top of memory, accessing data from address 0 at the bottom of memory. Normally, a 64kB ROM with initialization code is at the top of memory and RAM is at the bottom, as shown in Figure 4-41.

Figure 4-41. Typical Memory Use at Start of Execution



After the registers are initialized, the processor executes the instruction at FFFFFFF0h as its first operation. Normally, this operation will be a near jump in ROM to the start of initialization code elsewhere in the ROM. The 12 high-order address bits of the CS register remain set to 1 until one of the following occurs:

- The CS register is explicitly loaded with a segment selector.
- An inter-segment CALL, JMP, RET, or IRET instruction executes.
- An interrupt or exception occurs.

Any of these actions will reload the entire CS register and allow normal addressing to begin.

Real-Mode Initialization

Real-mode initialization only requires that interrupt handling routines be installed. This involves loading the routines in memory, loading the interrupt vector table (which starts at memory address 0), and enabling interrupts. Real mode does not use other tables, such as descriptor tables.

Because the nonmaskable interrupt (NMI) is always enabled, there is a short period of time between the end of reset and the completion of the NMI handling routine installation, during which an NMI would not be managed properly if it were to occur. The system hardware design must take this situation into account to prevent an NMI from occurring during that time. System software may therefore have to specifically enable the NMI after the NMI handling routine is installed. For example, IBM-compatible systems provide NMI hardware that is enabled through a write to I/O port 70h, bit 7.

Protected-Mode Initialization

Before switching to protected mode operation, the initialization code must establish a GDT in memory and load its base address and limit into the GDTR. At least two segment descriptors are required above the first (null) descriptor in this table: one for code and one for data. Before executing any instructions that use the stack, the stack pointer (SP) register must be initialized. The initialization stack can be simplified by making it part of the data segment, thereby eliminating the need for a separate stack segment and descriptor.

After the global descriptor table is established, the LGDT instruction is used to load the table's base address and limit into the GDTR. To prepare for interrupts, an IDT and an interrupt gate descriptor must be created. The LIDT instruction loads the IDT base address and limit to the IDTR.

The processor can then be switched to protected mode by setting the protection enable (PE) bit in CR0 to 1. To do this, the contents of CR0 must be read, the PE bit set, and the contents written back by means of the MOV CR0 instruction or the SMSW/LMSW instructions. The instruction immediately following this operation should be a JMP, which will flush the instruction queue.

At this point, the processor is operating in protected mode at the highest current privilege level (CPL = 0). The initialization code must reload all segment registers, which still contain their old real mode values, with values that are appropriate for protected mode.

Memory Segmentation

In protected mode, the memory management features that are implemented determine the types of data structures required. One GDT, with one code segment descriptor and one data segment descriptor, is always required. This flat memory model operates at the most basic level of segmentation.

A more flexible system uses multiple sets of such segments. The operating system itself will probably require multiple descriptors. Then, each task operating under it will require its own LDT, for which there must be a corresponding descriptor in the GDT. The operating system can allocate memory and assign new descriptors and descriptor tables as they are needed; or the initialization code can create them so that they remain as stable data structures.

Paging Mechanism

The initialization code can also enable paging. Before doing so, a page directory must be created and its base address must be loaded into the page descriptor base register (PDBR), the CR3 register. At least one second-level page table must also be created. Then the contents of CR0 can be read, the paging enable (PG) and protection enable (PE) bits can be set to 1, and the contents can be moved back to CR0.

Paging can be enabled in protected mode, if the page directory and page table have been installed. In any case, the instruction that sets or clears the PG and/or PE bits must be followed by a JMP instruction, which flushes the instruction queue. For proper operation, code that enables paging must exist in a region of memory that has the same physical address whether or not paging is enabled.

Multitasking Environment

To support multitasking, create a task state segment (TSS) and load its descriptor into the GDT, marking the descriptor as "not busy." Use the LTR instruction to load a segment selector for this TSS descriptor into the task register (TR). The LTR instruction will mark the descriptor as "busy" without performing a task switch. In this way, the first task switch that occurs will copy the current state into the TSS.

Use the LTR instruction once to prepare for the first task switch. After this, the processor's task-switching mechanism manages the "busy/not busy" status of the TSS descriptor. Either the operating system can create, assign, and deallocate TSSs dynamically, or they can be created by the initialization code and remain as stable data structures.

SuperState V Mode

SuperState V mode is a new and special extension of the Super386 microprocessor's architecture to provide OEMs with a method of creating product differentiation (e.g., power management and device emulation). A SuperState V program uses a separate address space called SuperSpace.

SuperState V mode allows a control program, running at a higher privilege level than the operating system, access to the Super386 processor for special system management and feature control purposes. SuperState V mode gives complete control of the processor to the system management code without the assistance, cooperation, or knowledge of the operating system.

In the 80386 processor, standard interrupts could be used for these system management functions, but since the operating system typically sets up the interrupt descriptor table (IDT), changes would be required to the operating system to gain its cooperation. In addition, code would have to be written specifically for each possible operating system. This would make the development costs of system management features prohibitive, not to mention the enormous costs of maintenance.

Using SuperState V mode, OEMs can build system management features into a system without having to interface with the operating system or change one line of operating system code. OEMs can also use SuperState V mode to implement simple multitasking between several operating systems, operating system independent disk caches, performance measurement tools, real-time diagnostic routines, virtual devices, or user-defined instructions.

SuperState V mode has direct access to many of the Super386 internal functions and registers. Much of the SuperState V application program is specific to the OEM's design. Consult the *SuperState V Architectural Manual* for specific information on writing SuperState V software.

In the remaining sections, Super386 modes will be referred to as either SuperState V mode or user mode. User modes are real mode, protected mode, and virtual-8086 mode.

Entering SuperState V Mode

SuperState V mode is entered in one of the following ways:

- Asserting the ANMI* pin (38605DXE processor only)
- Using the SCALL instruction
- Selecting one or more of the externally signaled interrupts
- Selecting one or more of the internally generated interrupts or exceptions
- Accessing a specific I/O port or a range of I/O ports
- Detecting a shutdown condition but before generation of a shutdown cycle
- Detecting an HLT instruction.

When the Super386 processor enters SuperState V mode, the processor reads a segment descriptor from memory at physical address 000FFFC0h or 000EFFC0h. The descriptor defines a region of memory where the user mode processor state will be saved and where SuperState V code resides.

SuperState V Segment Descriptor

The format of the SuperState V descriptor differs from the format of the descriptor in an LDT or the GDT. The SuperState V descriptor sets up a read/write data segment that is also an executable code segment.

In general, systems supporting SuperState V mode are constructed with memory subsystems that recognize the AADS* signal, which is generated only by the 38605DXE processor. When AADS* is not used, the SuperState V application programmer should initialize the SuperState V descriptor to point to a region that is available for SuperState V use within the normal user memory space.

Note: When SuperState V memory resides in user space, some of the system security capabilities that SuperState V offers are diminished.

The Super386 extended instructions and SuperState V code use a reserved area to save and restore portions of the CPU state. This area also may be used to contain pointers to I/O port and interrupt/exception capture tables.

Saved Information

When the Super386 processor enters SuperState V mode, it saves certain information into the SuperState V save area in order to free processor resources for use by SuperState V code. The Super386 processor state is restored when SuperState V mode is exited. To provide for a fast entry and exit from SuperState V mode, only a small subset of the processor state is saved initially. If a SuperState V application needs more registers than are initially saved, it must explicitly save and restore the additional registers.

Because only the code segment descriptor is saved, all references to SuperState V memory must use the CS: override prefix. If this is inconvenient, additional segment descriptors can be saved to free them for SuperState V code use. Once the information has been saved, the CS descriptor and EIP are loaded according to the means of SuperState V entry. For certain entries, EDX and EBX are also loaded with useful information.

SuperState V Entry Vectors

When the Super386 processor determines that it is to enter SuperState V mode, it loads the SuperState V descriptor; stores EIP, EFLAGS, EDX, EBX, and the CS descriptor in the SuperState V segment; fetches a vector corresponding to the cause of SuperState V entry; and begins execution at the location indicated by the vector.

When an IN, OUT, INS, or OUTS instruction causes SuperState V entry, the EDX register is loaded with the port number that the instruction was accessing, the port size is loaded into bits 31:8 EBX, and the instruction length (including any prefixes) is loaded into BL. The EIP saved on the stack points to the I/O instruction.

The I/O instruction is faulted before the instruction enters the execution unit. This means before an access to the I/O device is generated but after the processor has performed all protected mode privilege checks on the instruction. This I/O fault allows all operating system checks to be performed and any exceptions to be reported to the operating system before control is passed to the SuperState V program. The I/O fault also ensures that SuperState V mode is only invoked for those instructions that actually generate an I/O access.

When an interrupt or exception causes a SuperState V entry, the DX register is loaded with the zero-extended exception vector number. If an interrupt is caused by software, the BL register is loaded with the zero-extended instruction length. If an internally signaled interrupt causes a SuperState V entry, the instruction length value is unrelated and should be ignored.

Events, Ports, and Interrupt Capture (EPIC) Facility

The event capturing facility allows entry into SuperState V mode by the selection of specific port or interrupt vector ranges. The EPIC facility consists of seven Super386 registers that provide for six ranges of events, each of which can selectively capture either a single port range or a single interrupt range. Because the facility is implemented on the processor, it introduces no additional delay when enabled, unlike the I/O permission bit-map used by protected mode.

The EPIC facility can be operated in either inclusive or exclusive mode. In inclusive mode, SuperState V mode is entered when any match occurs between one of the six ranges and the corresponding operation. In exclusive mode, SuperState V mode is entered only when no range matches the corresponding operation. Control of the EPIC is discussed in the *SuperState V Architectural Manual.*

When a range of ports or interrupts/exceptions are matched, the logic ignores one or more of the least significant bits of the port or interrupt address. Matching 128 selected port numbers is accomplished by ignoring the least significant seven bits of the port address in the EPIC register. This means that a range of four ports can include port numbers 4 to 7 but not port numbers 2 to 5.

The EPIC facility should be operated in inclusive mode only when six or fewer devices are to be monitored and/or emulated by software. If more than six devices are required, the EPIC facility should be operated in exclusive mode. In exclusive mode, only performance critical operations need be placed in the EPIC registers when they are encountered. In this way, the EPIC facility can be operated much as a cache or TLB, providing unlimited capture capabilities with little or no observable performance loss.

SuperState V Programmer's Environment

A number of processor features cannot be used or have functions that are altered in the standard implementation of SuperState V mode. These features are discussed in the following paragraphs.

Cache—Any instruction, data, or unified cache present on the processor is disabled. Its contents are retained while in SuperState V mode, but no SuperState V code or data will be placed in it.

Segmentation Rules—Real mode segmentation rules are followed, with the exception that the default bit (D) can be set to 1, allowing for access to a full 32-bit address range.

Debugging—Debug exceptions are not generated.

Paging—Paging is disabled.

GS Segment—The GS: instruction prefix causes the associated memory reference to go to user mode memory space rather than SuperSpace. This provides a means for SuperState V code to examine user mode memory.

Hardware Interrupts—All hardware interrupts (INTR, NMI, and ANMI*) are masked.

Invalid Opcodes —The invalid opcode exception is disabled. In some cases, ordinarily undefined opcodes perform special functions specifically for SuperState V software. These special instructions are defined in the *SuperState V Architectural Manual.*

Software Interrupts—Software interrupts and exceptions require that an IDT be set up in SuperSpace. The use of the IDT implies real mode inter-segment transfers, which restrict addresses to the lower 1MB.

Execution Starting Address—Execution begins at an address specified during the SuperState V entry sequence. Execution may be defaulted to 16-bit code or 32-bit code, depending on the SuperState V segment descriptor, and can be altered in SuperState V mode. Switching from one mode to the other requires careful programming.

16-Byte Alignment of Segment Base Address

If inter-segment transfers occur in SuperState V mode, regardless of whether execution is in 16-bit or 32-bit mode, the base address of the SuperState V segment should be aligned to a 16-byte boundary, and it should be less than 1MB so that it can be expressed as a real mode segment. This is necessary because the real mode segmentation rules map selectors to base addresses by shifting left four bit positions.

The SCALL Instruction

The SCALL instruction is the only Super386 extended instruction available in user mode. It is used as a SuperState V procedure call where the source operand specifies the service requested of the SuperState V program. In some cases, the CPU may provide services directly without having to establish and initialize a SuperState V descriptor and its associated code.

The SCALL operations supported directly by the CPU include enabling and disabling any on-chip cache (the instruction cache in the case of the 38605); obtaining the CPU identification (family, version, silicon stepping level); and enabling SuperState V mode. Until SuperState V mode is enabled, any SCALL not handled directly by the CPU will return with the carry flag set to indicate that the service is not available.

System Security Issues

SuperState V mode provides mechanisms to circumvent operating system security. To prevent application programs from having access to SuperState V capabilities, the SCALL instruction can only be executed for enabling or disabling the cache, or for enabling SuperState V mode when the processor is at CPL zero (most privileged). Any other service request will return either CF = 1 if SuperState V mode is not enabled, or a value corresponding to the requested service if SuperState V mode is enabled.

To ensure system integrity, the SuperState V software must examine the CPL of the CPU upon each entry from the SCALL instruction and determine if the requesting program should be allowed access to the service it is requesting. For example, an application program should not be allowed to request that it be invoked each time a page fault occurs. This could cause serious performance problems.

SuperState V software allows system integrators to provide system BIOS level support for SuperState V mode while maintaining the integrity of protected operating systems such as UNIX. For system integrators that implement SuperState V support, operating systems like UNIX can retain their integrity but still access the features that the system manufacturer implemented in SuperState V mode.

Some applications may disable protection features while in SuperState V mode, and protection violations may be fatal. For this reason, a limit of 4GB (limit = FFFFFh, granularity = 1) on the SuperState V segment is recommended.

Instruction Pipeline and Cache Consistency

A cache-consistency mechanism is provided to ensure that instructions contained in the pipeline of both the 38600 and 38605 processors or in the instruction cache of the 38605 processor accurately reflect the contents of memory. Because instructions can be present in the pipeline without being present in the cache, both the 38600 and 38605 processors contain identical consistency checking hardware. This hardware functions on the 38605 processor whether the cache is enabled or not.

The mechanism keeps a record of instructions contained in the cache or pipeline. When a store is executed to an address that matches one recorded by the mechanism, the instruction pipeline is flushed. External devices that store to memory located in the instruction cache of the 38605 processor will also cause the corresponding cache entry to be invalidated.

Instruction Cache (38605 Only)

The 512-byte instruction cache in the 38605 processor increases processor performance for most operations. It contains 32 directly mapped 16-byte entries, each of which has tag information allowing it to map to any address value for bits 31 to 9. Each four bytes contains a valid bit, allowing for partial validity of each 16-byte entry. When instruction data is available from the cache, the external bus is available for operand accesses. Four bytes can be read from the cache in a single cycle, or eight bytes in the equivalent of one bus access. Special hardware is also included to generate addresses for jump instructions.

In some cases, this combination of cache and hardware address generation dramatically increases execution speed. On average, about 65 percent of all instruction fetches are satisfied by the cache. The actual cache hit rate varies dramatically, however, from zero to 100 percent, depending on the nature of the executing code.

The 38605 processor can be operated with the instruction cache enabled or disabled. The assertion of the KEN* signal enables the cache on each instruction fetch. When asserted, the code fetch is written into the cache and the entry is made valid. Future accesses to the same address will retrieve the data from the cache. Software cannot depend on the contents of the cache being retained while it is disabled. Similarly, software cannot assume that the entire contents of the cache is invalidated by the act of disabling it.

To invalidate the entire cache, the FLUSH* signal must be asserted. Invalidating the cache from software is unnecessary because the consistency mechanism ensures that the cache always reflects an exact copy of what is in memory.

Shutdown and Halt

A shutdown occurs if a fault is raised during the servicing of a double-fault exception. A halt occurs when a HLT instruction is executed. In either case, the processor enters a halt cycle, in which it performs the following actions in the sequence shown:

1. Stops executing instructions.
2. Places one of two addresses on the address bus:
   00000000 = HLT instruction was executed.
   00000002 = a shutdown (fault on double-fault) occurred.
3. Releases any locked resources.
4. Waits for an external interrupt.

As with normal interrupts, a nonmaskable interrupt or reset is needed if maskable interrupts are disabled. After the interrupt signal is received, the processor will service it normally, return control, and continue execution. If a halt occurred, execution continues at the instruction that follows the HLT instruction. If a shutdown occurred, execution continues from an uncertain point.

Testing the TLB

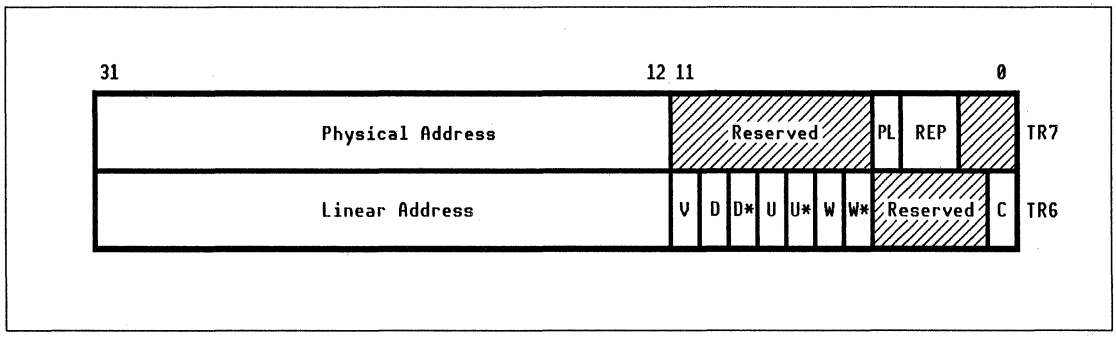
The structure and function of the translation lookaside buffer (TLB) is described in the "Paging" section of this chapter. While it is very unlikely that the TLB will fail, two registers are provided for TLB testing:

- Test data register (TR7)—Holds a physical address and attributes.
- Test command register (TR6)—Holds a corresponding linear address and attributes.

These registers, illustrated in Figure 4-42, can be used to write and read TLB entries in a power-on self-test routine or at other times. The MOV instructions are used to load and store the registers. In real mode, the MOV instructions are always available. In protected mode, the MOV instructions are valid only when executed at the highest current privilege level (CPL = 0). The instructions cause a general-protection exception if used at a less privileged level.

Note: Paging must be disabled before TLB testing begins.

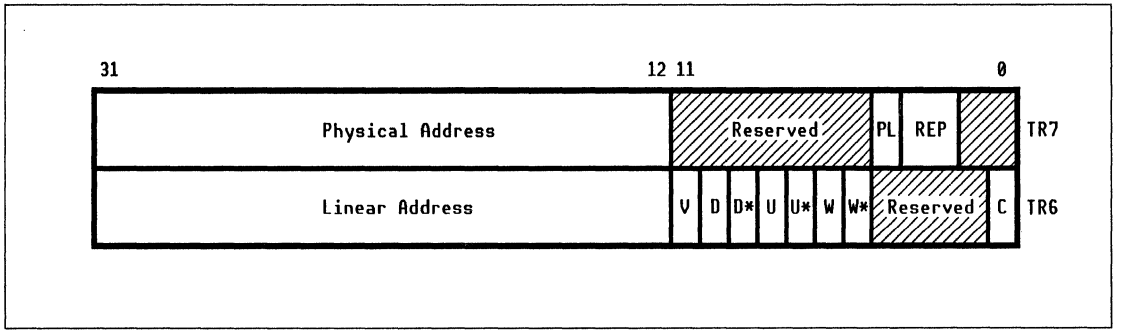
Figure 4-42. *Test Registers TR7 and TR6*



Writing TLB Entries

To write TLB test entries, a physical address is first moved to TR7. Figure 4-43 shows the TR7 register setting. The pointer location (PL) must be 1, and the replacement (REP) field must specify which of the four associative data blocks (called *ways*) are to hold the address.

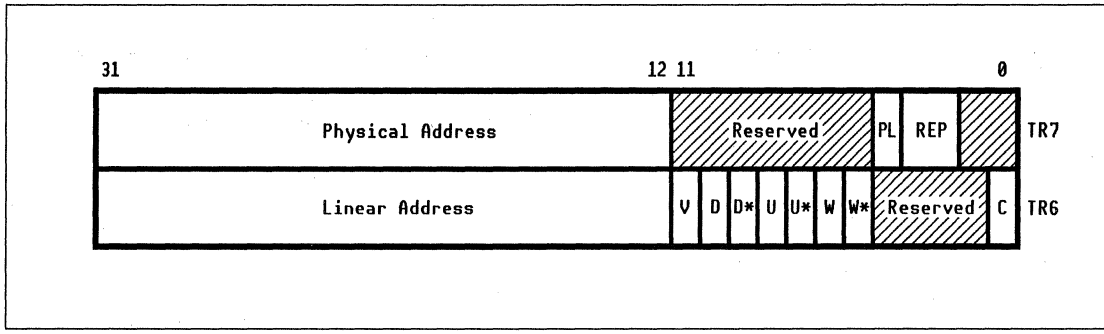
Figure 4-43. *TR7 Register Settings for Writing a TLB Entry*



31:12		Physical Address—For a TLB write, these bits specify the physical address that corresponds to the linear address specified in the TR6 register (Figure 4-44).
4	PL	Pointer Location—For a TLB write, this bit is set to 1 if the REP bits select the associative data blocks for the entry. If this bit is clear, the internal pointer of the paging unit indicates the selection: 1 Use REP bits to select block. 0 Internal pointer selects associative block.
3:2	REP	Replacement—For a TLB write with PL = 1, these bits determines which of the four associative data blocks are to hold the TLB entry being written. If PL = 0, these bits have no meaning. 11 Way3 10 Way2 01 Way1 00 Way0

After TR7 is written, TR6 is written with the corresponding linear address, the attribute bits, and the C bit = 0. Figure 4-44 shows the TR6 register settings.

Figure 4-44. TR6 Register Settings for Writing a TLB Entry



31:12		Linear Address—For a TLB write, these bits specify the linear address that corresponds to the physical address, already written to TR7, in the TLB entry (see Figure 4-44).
11	V	Valid Data—For a TLB write, this bit indicates whether the TLB entry contains valid data. When testing the TLB, this bit is set to 1; otherwise the entry will be deemed invalid if found on a TLB search. Writing to register CR3 clears the V bit in all TLB entries. 1 Valid 0 Invalid.
10:9	D, D*	Dirty Attribute Bit and Its Complement—For a TLB write, these bits affect the setting of the D bit in the TLB entry tag. The bit-pair meanings are:

D	D*	Meaning
0	0	Setting not defined.
0	1	Clear to D = 0.
1	0	Set to D = 1.
1	1	Setting not defined.

8:7 U, U*

User/Supervisor Attribute Bit and Its Complement—The U bits are also called the U/S bits. For a TLB write, these bits affect the setting of the U bit in the TLB entry tag.

U	U*	Meaning
0	0	Setting not defined.
0	1	Clear to U = 0.
1	0	Set to U = 1.
1	1	Setting not defined.

6:5 W, W*

Read/Write Attribute Bit and Its Complement—The W bits are also called the R/W bits. For a TLB write, these bits affect the setting of the W bit in the TLB entry tag.

W	W*	Meaning
0	0	Setting not defined.
0	1	Clear to W = 0.
1	0	Set to W = 1.
1	1	Setting not defined.

0 C

Command—For a TLB write, this bit must be cleared.

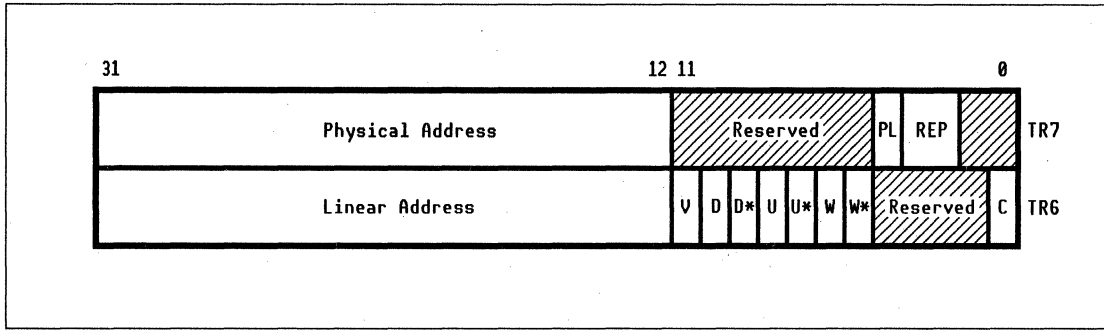
1	Search the TLB.
0	Write to the TLB.

Reading TLB Entries

In reading TLB entries, called a TLB *search* or *lookup*, the TR7 register returns the physical address that corresponds to the linear address in the TR6 register. The operation begins by moving a linear address to TR6, with the attribute bits set as described following Figure 4-45 and with the C bit set to 1. Then TR7 is read. If the pointer location (PL) in TR7 is set to 1, this indicates that the read was successful and the corresponding physical address and REP field (indicating the associative data block or way) can be read.

The TR6 segment settings for searching and after searching the TLB are shown in Figure 4-45.

Figure 4-45. *TR6 Settings for Searching and After Searching the TLB*



31:12		Linear Address—On an entry search, the TLB is searched for this 20-bit value. If a unique match is found, the entry is returned in TR6 and TR7.
11	V	Valid Data—After returning from a successful TLB entry search, this bit indicates that the V bit of the TLB entry is set to 1 (valid). 1 Valid 0 Invalid

10:9 D, D*

Dirty Attribute Bit and Its Complement—For a TLB entry search, these bits set conditions as shown below. After the search, these bits reflect the comparable bit settings found in the TLB entry.

D	D*	Meaning
0	0	Find no matches.
0	1	Find if D = 0.
1	0	Find if D = 1.
1	1	Find all, ignoring D.

8:7 U, U*

User/Supervisor Attribute Bit and Its Complement—The U bits are also called the U/S bits. For a TLB entry search, these bits set conditions as shown below. After the search, these bits reflect the comparable bit settings found in the TLB entry.

U	U*	Meaning
0	0	Find no matches.
0	1	Find if U = 0.
1	0	Find if U = 1.
1	1	Find all, ignoring U.

6:5 W, W*

Read/Write Attribute Bit and Its Complement—The W bits are also called the R/W bits. For a TLB entry search, these bits set conditions as shown below. After the search, these bits reflect the comparable bit settings found in the TLB entry.

W	W*	Meaning
0	0	Find no matches.
0	1	Find if W = 0.
1	0	Find if W = 1.
1	1	Find all, ignoring W.

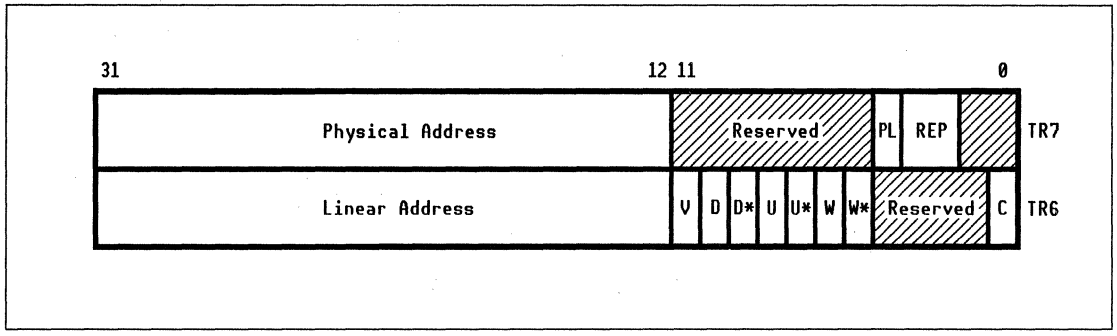
0 C

Command—For a TLB entry search, this bit must be set to 1.

1	Search the TLB.
1	Write to the TLB.

The TR7 segment in Figure 4-46 shows the values returned after the search.

Figure 4-46. TR7 Return Values After TLB Search



31:12		Physical Address—After a TLB search, this field returns the physical address that corresponds to the linear address specified in TR6.
4	PL	Lookup—After a TLB search, this bit indicates whether the search was successful or not. 1 Match found in TLB. 0 No match.
1:0 3:2	REP	Report—After a TLB search in which PL = 1 indicates that a match was found, these bits indicate which of the four associative data blocks contained the tag that was found. If PL = 0, these bits have no meaning. 11 Way3 10 Way2 01 Way1 00 Way0

Debugging

A set of debug registers is provided to assist debug programs. To use debugging, the debug registers are loaded with the memory addresses whose access should cause program execution to stop. These addresses are called *breakpoints*. They can refer to either instruction or data locations. When a breakpoint is encountered, the processor generates a debug exception so that a debug-exception handling routine can service the event.

Traditional Debugging With Interrupt Vector 03

In the traditional method of setting breakpoints, without debug registers, the instruction opcode at the breakpoint in memory is replaced by the INT 03 opcode. When the INT 03 opcode is encountered, control is transferred to the breakpoint trap handler for interrupt vector 3, which should maintain a copy of the original instruction in memory. To resume program execution, the interrupt routine can then execute the substituted instruction, restore it to its place in memory (at the breakpoint), and perform an IRET to continue execution at the substituted instruction in memory.

Breakpoints can be implemented more simply with the processor's breakpoint registers, as described below. However, the INT 03 vector is still reserved for this traditional type of debug routine and can be useful when more than four breakpoints are desired.

Using the Debug Registers

Breakpoint addresses can be entered directly into these registers; instructions in memory need not be substituted with INT 03 opcodes. The registers also allow breakpoints to be set in ROM, which is not possible in the traditional debugging method. Figure 4-47 illustrates the eight 32-bit debug registers, two of which are reserved. The MOV instruction is used to load and store the registers.

Figure 4-47. Debug Register Set

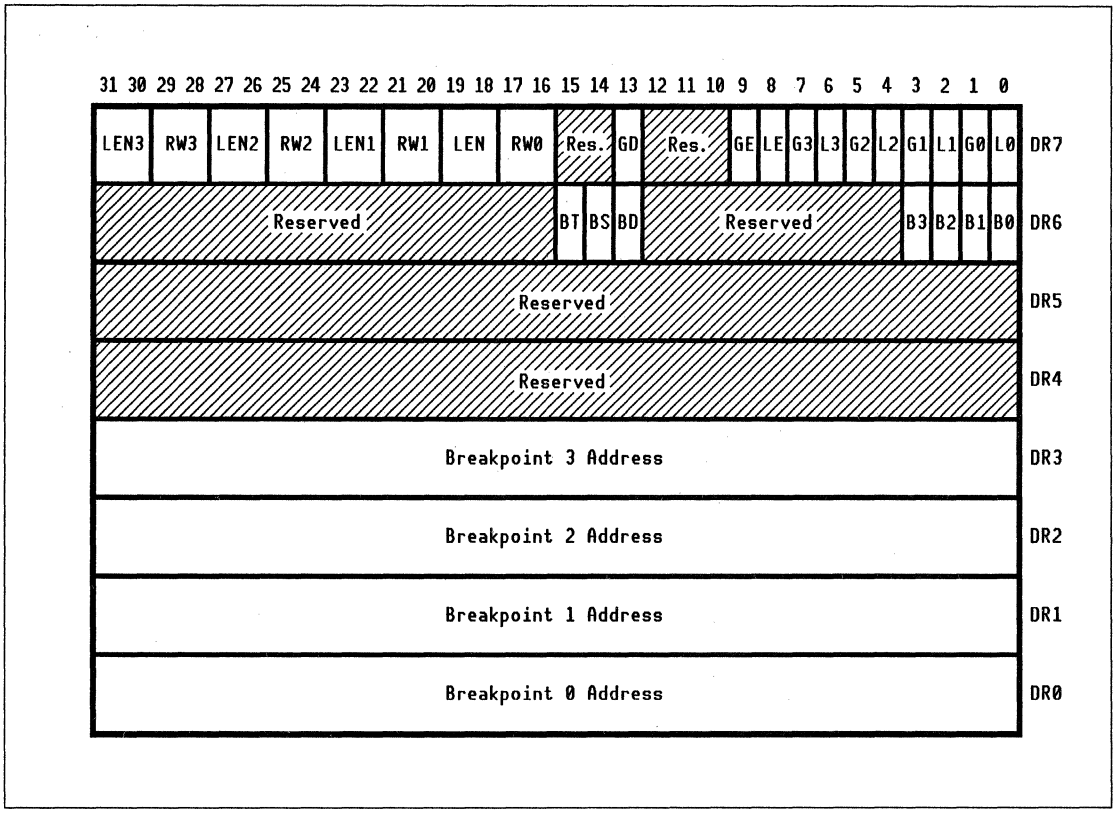


Table 4-15 shows the functions of the debug registers illustrated in Figure 4-47.

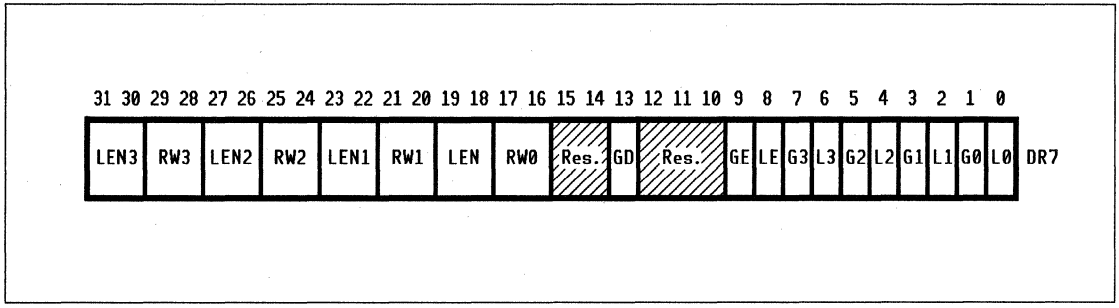
Table 4-15. Debug Register Functions

Register	Description
DR7	Debug Control. This register determines the behavior of the four breakpoint registers, DR3:0. See the "Debug Control and Status" section.
DR6	Debug Status. When a debug exception has occurred, the register containing the breakpoint and other information about the exception is returned in this register. See the "Debug Control and Status" section.
DR5:4	Reserved
DR3:0	Breakpoints. Up to four breakpoint addresses in linear memory can be specified in these registers. The conditions under which each breakpoint will be valid are controlled through DR7.

Debug Control and Status

Debug control register DR7 defines the type of access that will cause a debug exception to be generated at each breakpoint address. When a debug exception occurs, debug status register DR6 can be read to determine how it occurred. The list following Figure 4-48 shows the organization of the DR7 register.

Figure 4-48. Debug Control Register DR7

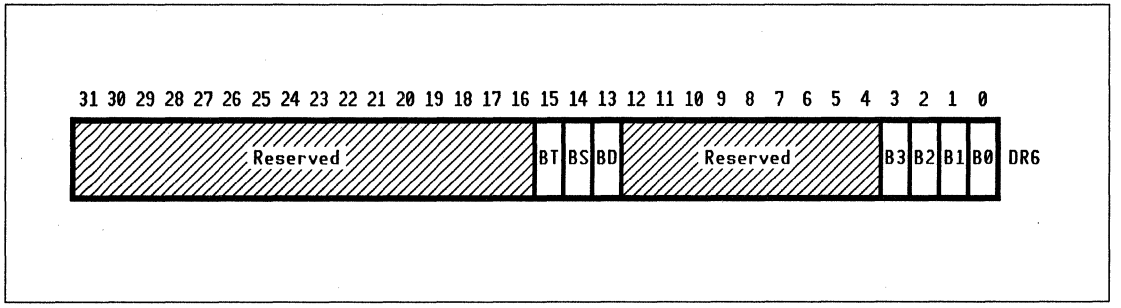


bits:	31:30 27:26 23:22 19:18	LEN3 LEN2 LEN1 LEN0	Length of Breakpoint—These bit pairs select the length of the data pointed to by the DR3:0 breakpoint address registers. The bit pairs correspond to the following address lengths: 11 Four bytes 10 Reserved 01 Two bytes 00 One byte. If the corresponding RW bit pair for a breakpoint indicates instruction execution (00), the LEN bit pair must be cleared to 0.
	29:28 25:24 21:20 17:16	RW3 RW2 RW1 RW0	Read/Write Break Condition—These bit pairs specify a condition under which a break will occur for the opcode or data that is pointed to by the DR3:0 breakpoint address registers. The bit pairs correspond to the following break conditions: 11 Break on data read or write. 10 Reserved. 01 Break on data write. 00 Break on instruction execution.
	13	GD	Global Debug Access Detect—This bit controls whether the BD bit (bit 13) of the DR6 will reflect read/write access attempts to any of the debug registers DR0:7 while they are in use by in-circuit emulation. 1 Enable access detection. 0 Disable access detection.

9	GE	Global Breakpoint on Exact Match—On the Super386 processor, this bit has no effect. All matches are exact, and execution overlapping is never disabled in order to achieve this. The bit is defined here only for compatibility with the 80386 architecture. The bit is cleared on a task switch.
8	LE	Local Breakpoint on Exact Match—On the Super386 processor, this bit has no effect. All matches are exact, and execution overlapping is never disabled in order to achieve this. The bit is defined here only for compatibility with the 80386 architecture. The bit is cleared on a task switch.
7 5 3 1	G3 G2 G1 G0	Global Breakpoint Enable—These bits enable the corresponding breakpoint in the DR3:0 registers, on an ongoing basis. The processor does not clear these bits when it switches to a new task. 1 Enable breakpoint for all tasks. 0 Disable breakpoint.
6 4 2 0	L3 L2 L1 L0	Local Breakpoint Enable—These bits enable the corresponding breakpoint in the DR3:0 registers for a single task only. The processor clears these bits when it switches to a new task. 1 Enable breakpoint for this task only. 0 Disable breakpoint.

Register DR6 returns information that was valid at the time the debug exception was generated, allowing the debug handler to determine the reason for the exception. The processor does not clear the contents of DR6. The register should be cleared by the debug handling routine to avoid confusion on the next debug exception. The list following Figure 4-49 gives the organization of DR6.

Figure 4-49. Debug Register DR6



bits:	15	BT	Breakpoint Trap—This bit is set to 1 if the debug exception was generated when the processor switched to the current task and found that the debug trap bit (bit T) of the TSS was set to 1. 1 TSS trap bit was set during task switch. 0 Not a task-switch exception.
	14	BS	Breakpoint Single-Step—This bit is set to 1 if the debug exception was generated because the trap flag (TF) in the EFLAGS register was set to 1. 1 Single-step trap after instruction execution. 0 Not a single-step debug exception.
	13	BD	Breakpoint Debug—This bit is set to 1 if the next instruction would perform a read or write access on one of the debug registers DR0:7 while they are in use by in-circuit emulation. 1 Next instruction accesses one of DR0:7. 0 Next instruction not debug.
	3 2 1 0	B3 B2 B1 B0	Breakpoint at Breakpoint Address—One or more of these bits are set if the address in the corresponding breakpoint address register DR3:0 could have caused the debug exception. The bits will be set as long as the conditions specified by the corresponding LEN and R/W bits are met, and will be set regardless of the L3:L0 and G3:0 settings. 1 Breakpoint at DR3 address 1 Breakpoint at DR2 address 1 Breakpoint at DR1 address 1 Breakpoint at DR0 address.

Conditions for Recognizing Breakpoints

Breakpoints that are set on instructions (RW bits = 00 in the debug control register) must point to the first byte of the instruction opcode, or to the first prefix byte if the instruction includes any prefixes. Breakpoint operation is unpredictable if the LEN3:0 field is set to anything other than 00 for instruction breakpoints.

Breakpoints set on data (RW bits = x1 in the debug control register) must specify the data size being accessed through the LEN3:0 field, which the processor uses to mask out the low-order address bits in the DR0:3 registers. For this reason, only data accesses on aligned boundaries (e.g., byte accesses on any boundary, 16-bit accesses on word boundaries, and 32-bit accesses on dword boundaries) generate useful results. Access to any of the bytes in the range specified by the LEN3:0 field will generate a debug exception. If a breakpoint must be set on misaligned data, two breakpoint addresses can be set to the adjacent byte locations.

Interrupt Vector 01

The processor reserves INT 01 for handling the debug exception. The exception handling routine should first check the debug status register to determine what type of debug exception occurred, as described in the following sections. A debug exception that is generated upon encountering an instruction to be executed is a fault, because the exception is generated before the instruction is executed. Debug exceptions generated for any other reason are traps, because the exception is generated after the instruction has executed.

Task-Switch Trap

When the program has transferred control to a new task, the processor checks the trap bit (T bit) of the new task's TSS. If the T bit is set and a debug exception occurs, the BT bit will be set in the DR6 register.

A conflict occurs if the debug exception handling routine is itself a task and its T bit is set. Trapping a task switch elsewhere will generate a debug exception, but transferring control to the debug exception handling routine will cause generation of another debug exception, starting an infinite loop.

Single-Step Trap

When the trap flag (TF) bit is set in the EFLAGS register, a debug exception occurs at the end of the current instruction execution. This is not true if the instruction is one that sets the TF flag, or if switching to a new task causes the TF flag to be set when EFLAGS is loaded. In both cases, the trap occurs on the next instruction. When the exception occurs, the processor clears TF and then transfers control to the debug exception handling routine. The EFLAGS image on the stack can be used to determine whether single-step execution should continue.

Single-stepping has a higher priority than INTR, so if they both occur at the same time, single-stepping occurs first. The single-step handler may clear the IF flag (through an interrupt gate, for example), preventing the INTR interrupt from occurring until the IF flag is once again set. This precedence ensures that an external interrupt will not be handled in single-step mode. All INTs clear the single-step (TF) flag. To single-step an external interrupt, an INT 03 or debug register must be used.

Developers of software debugging tools should note that the INT and INTO instructions cause TF to be cleared. The debugger must detect these instructions and replace them with equivalent code to effect the same transfer of control without actually executing the INT or INTO instructions.

General-Detect Fault

If the debug registers are used by in-circuit emulation, a conflict could occur if an instruction were to attempt access to the registers. To detect this type of interference, the debug exception handling routine can check the state of the breakpoint debug (BD) bit in debug register DR6. This bit is set to 1 if the next instruction would perform a read or write access on one of the debug registers DR0:7 while they are in use by in-circuit emulation.

Breakpoint Fault on Instruction Fetch

If an instruction is encountered at a valid breakpoint address, a debug exception is generated before the instruction is executed. The resume flag (RF) in the EFLAGS register can be used by the debug exception handling routine to restart instructions that cause non-debug faults. The handling routine simply sets to 1 the RF bit in the EFLAGS copy that has been pushed onto the stack local to the routine. In this way, resuming execution at the same breakpoint address will not generate additional debug exceptions due to breakpoint faults. Moreover, other exceptions such as breakpoint traps and non-breakpoint faults will continue to be serviced.

Certain Instructions

IRET and POPF—plus JMP, CALL, or INT instructions that cause a task switch—change the RF flag according to its saved value in the EFLAGS register. Except for these instructions, the processor always clears RF when an instruction successfully completes. If the debug handling routine were to retry a faulting instruction after a debug fault, the instruction could also cause other faults. Each time the instruction is restarted after these other faults, RF remains set to 1. Continued debug faults are avoided until the instruction successfully completes and clears RF to 0.

Before executing a fault handling routine, the processor sets RF to 1 in the EFLAGS copy that has been pushed onto the stack. In this way, the instruction that restores EFLAGS values (such as RF) before returning from the routine will again set RF and allow execution to resume at the same breakpointed instruction. No repeated exceptions will be generated for the same instruction.

Breakpoint Trap On Data Access

The breakpoint registers allow data locations in memory to be monitored for activity. When the processor has executed an instruction that accesses data at a valid breakpoint address, it generates a debug exception. The debug exception handling routine can immediately determine the data access that occurred.

Even if the processor is starting to execute the next instruction by the time the trap occurs on the current memory access, the instruction trapped by the exception will always be the current instruction, not the next one in the processing queue. On the Super386 processor, the GE and LE bits in debug register DR7 have no effect. All matches are exact, and execution overlapping is never disabled in order to achieve this. The bit is defined only for compatibility with the 80386 architecture.

Because the processor completes instruction execution before generating a debug exception, the data access being trapped has already occurred when the exception handling routine sees the data. Therefore, debugging software might have to make a copy of any necessary breakpointed data, in case the trapped access happens to overwrite the data of interest.

Single-stepping a HLT instruction normally causes the single-step event to occur after the halt state is exited, due to a pending interrupt. This is not the case. Instead, a single step is taken immediately. A debugger must be aware that the instruction following the halt should not be allowed to execute until a pending interrupt arrives.

Because interrupts can occur before execution of an instruction with the RF flag set, the RF flag may not function as expected when executing 16-bit code. The interrupt will occur in this case, pushing only 16 bits of the EFLAGS register on the stack. After the interrupt handler clears RF and returns to the original program, FLAGS will be restored but RF will not, because it is in the upper 16 bits. As a result, multiple instruction debug faults will occur.

Operand debug events in repeated string instructions are recognized after the string iteration that matched the debug address completes, and before any further iterations. If this event is not on the last iteration, the EIP saved on the stack will point to the repeated string instruction. If this event is on the last iteration, the EIP will point at the next instruction.

Other Processor Modes

The previous sections of this manual have concentrated on protected mode, the processor's native execution mode. The following sections discuss real mode and virtual-8086 mode, which are designed to accommodate programs written for 16-bit processors like the 8086 and 80286.

Protected mode is the protected virtual-address mode in which all of the processor's segmentation (memory-protection) and/or paging (virtual-address) capabilities are available. Programs written for protected mode on the 80386 and 80286 processors can be run in protected mode on a Super386 processor, because 80386 and 80286 code are subsets of Super386 code. Maximum linear memory size is 4GB.

In real mode, the 8086 real-address emulation mode, none of the protected-mode segmentation or paging functions are available. The processor is initialized to this mode upon power-up or reset. Maximum memory size (1MB), default operand size (16 bits), address generation, and interrupt handling are similar to the 80286 real mode. Instruction prefixes allow use of 32-bit operands, giving full use of the 32-bit registers. All code runs at privilege level 0.

In virtual-8086 mode, the processor generates addresses as in real mode, but with the paging capabilities of protected mode. This is a sub-mode of protected mode. Virtual-8086 mode allows you to run programs written for the 8086 processor as a task on the Super386 processor. Like real mode, virtual-8086 mode has a maximum memory size of 1MB. Instruction prefixes allow use of 32-bit operands, giving full use of the 32-bit registers. Under the control of system software, the processor can enter virtual-8086 mode from protected mode, run a 16-bit program, and then return to protected mode with no effect on protected code and data. Virtual-8086 programs run at privilege level 3. The Super386 protected-mode code that runs the virtual-8086 task executes at privilege level 0.

Real Mode

This mode is selected when protection is disabled with the PE flag in the CR0 register. In this mode, the processor performs similarly to an 8086 processor, except for the features and parameters described in this section. All segmentation protection features are turned off. There is no task switching, and the protection level is 0. All operands and addresses use the lower 16 bits of the general registers described in Chapter 2, "Programmer's Model." Interrupts and exceptions are treated differently than they are in protected mode. See "Interrupts and Exceptions" in this chapter for the discussion.

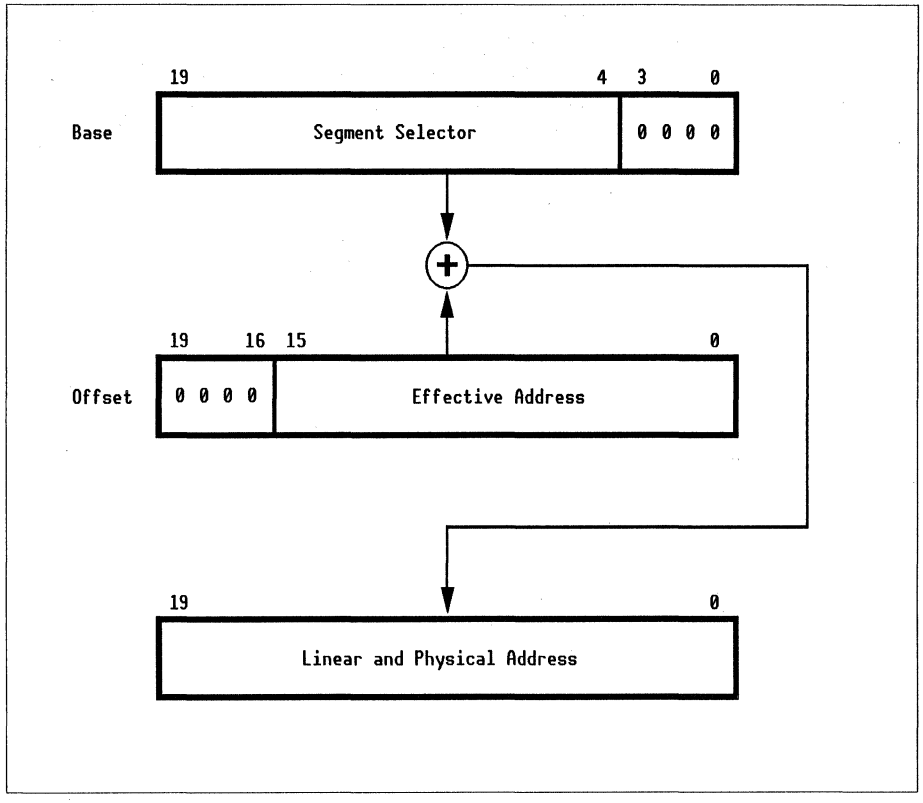
Address Formation

In real mode, the Super386 processor derives linear addresses in the same way as the 8086 processor. The 16-bit segment base address is shifted left by four bits (multiplied by 16), resulting in a 20-bit value. This value is added to a 16-bit offset to give a 20-bit linear address, which is also the physical address, for a memory space of 1MB. See Figure 4-50.

The default (D) bit (bit 22) in the CS segment descriptor's shadow register is always 0 in real mode. This means that the default address and operand size is 16 bits. An instruction prefix can override the default for operands or addresses, but the 32-bit address cannot exceed the 64kB segment size or an exception is generated. The address size attribute is automatically cleared to 0 following a system reset or initialization. This attribute is under explicit user control in protected mode.

Segment descriptors are not used. All segments have a maximum size of 64kB and a descriptor privilege level (DPL) of 0. A segment can start at any 16-byte boundary within the linear address space. No exceptions are generated when a segment register is loaded, because all possible values are valid.

Figure 4-50. Real-Mode Linear Address Generation



Address Limits and Boundary Crossing

Addresses higher than the 64kB limit generate a general protection exception (INT 13) or a stack fault exception (INT 12). This is unlike the 8086 processor, in which the address wraps around and no indication is given.

Instructions

All instructions operate in real mode except the following instructions, which are used specifically in protected mode or in multitasking:

ARPL	LSL	STR
LAR	LTR	VERR
LLDT	SLDT	VERW

The Super386 processor limits the length of instructions to 15 bytes. A general-protection exception caused by a long instruction usually indicates redundant prefixes. Unlike the Super386 processor, the 8086 processor does not generate an exception when the instruction exceeds 15 bytes.

The PUSH SP instruction works differently on the Super386 processor than on the 8086 processor. The Super386 processor pushes the stack pointer onto the stack before, not after, the SP register is incremented in the push operation. For this reason, PUSH SP instructions on the 8086 must be changed to the following:

```
PUSH BP
MOV BP,SP
XCHG BP, [BP]
```

In the 8086 processor, a PUSHF instruction sets bits 15:12 of the FLAGS register (the NT and IOPL flags, plus a reserved bit). In the Super386 processor, bit 15 is cleared to 0 (reserved), and bits 14:12 (the NT and IOPL flags) have the current value unchanged.

There are also differences in the operation of the DIV and IDIV instructions. In the Super386 processor, the instruction pointer points to the failed instruction, whereas the 8086 instruction pointer points to the next instruction. The Super386 processor generates the largest negative quotient for 80h or 8000h; the 8086 processor generates a divide-by-zero error (exception 0).

The LOCK Prefix

The LOCK prefix should only be used to protect data operations from interruption by another bus master. In hardware system designs that observe the Super386 architecture, a locked instruction will lock only the memory area designated by the destination operand. In the 80286 and 8086 architectures, the LOCK prefix causes the entire physical memory space to be locked. See Appendix C for more details.

Use LOCK only with the following instructions, when these instructions write to memory:

ADC	BTS	OR
ADD	DEC	SBB
AND	INC	SUB
BTC	NEG	XCHG
BTR	NOT	XOR

An invalid opcode (exception 6) will result from using LOCK with other instructions or with these instructions when they do not write to memory.

Undefined Opcodes

The Super386 instruction set is a superset of the 8086 instruction set. Some 8086 undefined opcodes are valid on the Super386 processor. For opcodes undefined on both processors, the Super386 processor generates an invalid opcode error (exception 6).

Interrupts and Exceptions

In real mode, interrupts and exceptions are handled the same way the 8086 processor handles them. The interrupt vector table contains dword entries that point to the interrupt handler. The low-order 16 bits of this pointer are an offset (which is the instruction pointer for the beginning of the handler), and the high-order 16 bits are a segment selector for the code segment.

All interrupt and exception vectors generated in protected mode are also generated in real mode. For a complete list of the vectors, see the section entitled "Summary of Interrupt and Exception Conditions." The LIDT instruction, which loads the IDTR, sets the base and limit of the interrupt vector table. The base should always be an address 0. A double fault (exception 8) will occur if an interrupt tries to use a vector outside the interrupt vector table.

When the Super386 processor is executing a nonmaskable interrupt, all other NMIs are masked until an IRET instruction is executed.

On the 8086 processor, if an instruction accesses a memory operand beyond the 64kB maximum offset permitted, or if it accesses a memory operand with an offset of zero, the instruction wraps around the boundary and does not generate an exception. On the Super386 processor, instructions that cross offsets 0 or 64kB generate a general protection exception (or a stack exception if a stack segment is addressed). If a series of instructions pass the 64kB maximum offset, the 8086 processor retrieves the next byte of the instruction from the zero offset location. The Super386 processor, on the other hand, generates a general-protection exception.

The 8086 external interrupt handler cannot be used for single-step operations when an interrupt occurs. The Super386 processor will single-step through an interrupt because the single-step interrupt has a higher priority than other interrupts.

Interfacing with a Coprocessor

When a coprocessor is installed, it must use the coprocessor error exception (exception 16) when an error occurs. Code written for the 8086 may use another exception in responding to an 8087 coprocessor error, but any such exception vectors should call the coprocessor error exception handler.

Entering and Leaving Real Mode

The processor enters real mode following a reset or power up. When it does so, the paging (PG) bit (31) in CR0 is automatically cleared to 0 to disable paging. The section entitled "Initialization" describes the initial state in detail.

Switching From Protected Mode to Real Mode

There are two ways to enter real mode from protected mode:

- Reset the processor from external hardware.
- Clear the PE bit.

The second method, clearing the PE bit, can be done in the following sequence:

1. Disable all interrupts.
2. Execute in a code segment that has the same address in both physical and linear addresses, and use data that also has the same physical and linear addresses.
3. Load the DS, SS, ES, FS, and GS segment registers with selectors for a read/write expand-up data segment of 64kB, using DPL = 0.
4. Clear the PG bit in CR0 to disable paging, and clear the PE bit to disable protection.
5. Execute a direct inter-segment JMP to flush the processor pipeline and transfer to the real mode program in the lower megabyte of physical memory.

Step 4 can be done using the following commands:

```
MOV reg,CR0
AND reg,7FFFFFFEh
MOV CR0,reg
```

Switching From Real Mode to Protected Mode

To leave real mode and return to protected mode, follow this procedure:

1. Disable all interrupts.
2. Load the GDTR and IDTR with the base and limit addresses of the GDT and IDT. The IDT must be in the format used by protected mode interrupts.
3. Initialize the GDT, IDT, TSS, and LDT.
4. Set the PE bit to 1 to enable protection.
5. Reload the CS register with an inter-segment jump. This flushes the execution pipeline of instructions fetched and decoded in real mode. It may be a jump to the next instruction.
6. Reload the segment registers with valid protected-mode selectors (or null selectors).
7. Load the TR register with a TSS selector.
8. Load the LDTR with a null selector or system segment of the LDT type.

Virtual-8086 Mode

In virtual-8086 mode, the processor generates addresses as in real mode, but with the paging capabilities of protected mode. This is a sub-mode of protected mode. In virtual-8086 mode, programs written for the 8086, 8088, 80186 or 80188 processor can run as a task on the Super386 processor. Like real mode, it has a maximum memory size of 1MB. Instruction prefixes allow use of 32-bit operands, giving full use of the 32-bit registers. Under the control of system software, the processor can enter virtual-8086 mode from protected mode, run a 16-bit program, then return to protected mode with the assurance that protected code and data have not been affected. Virtual-8086 programs run at privilege level 3, while Super386 protected mode system code runs at privilege level 0, 1, or 2.

Determining Addresses

The processor determines linear addresses by shifting the base address in the segment selector four bits to the left to form a 20-bit address. These two values are then added to create a linear address in the task's address space between 0 and 10FFEFh. Only the low-order 20 bits are mapped with page tables to a 32-bit physical address. See Figure 4-50.

The Super386 processor can also generate a 32-bit effective address using the address-size instruction prefix. However, the value of the address cannot exceed 65,535; if it does, an INT 12 or INT 13 interrupt will occur.

Entering Virtual-8086 Mode

Virtual-8086 mode is entered when a task switch loads the EFLAGS register with the virtual mode (VM) bit set to 1. Mode transfers can take place in one of the following ways:

- A task switch can load the EFLAGS register.
- An IRET instruction can load the new EFLAGS register from the stack.

The VM bit can only be changed when the current privilege level is 0. If a task switch is done, the new task must use a Super386 TSS. The VM bit is in bit position 17, which does not exist in the 80286 FLAGS register.

Exiting Virtual-8086 Mode

Virtual-8086 mode can be exited through an interrupt or trap gate, or by using an interrupt or exception to force a task switch. The new TSS loads the EFLAGS register with VM = 0 and executes the program under that TSS. You can also exit virtual-8086 mode with an interrupt or exception that vectors to a program at privilege level 0. This causes the processor to store current values of the EFLAGS register on the stack, then clear the VM bit.

Instruction and Register Usage

Virtual-8086 mode programs can execute programs containing 80186, 80188, 80286, and Super386 instructions. Instruction prefixes can be used for instructions with 32-bit operands. Unlike the 8086, two additional segment registers exist on the Super386 processor: FS and GS. They act just like the DS register, but they are never used by default for any operation. They must be requested explicitly with the FS and GS segment overrides. If these segments are referenced in a program run on an 8086 processor, an illegal opcode exception will be generated.

Instructions

Several considerations apply to the use of instructions in the virtual-8086 mode. These are discussed in the following paragraphs.

Instruction-Length Exceptions

The 8086 processor does not generate an exception if the instruction exceeds 15 bytes. The Super386 processor generates a general-protection exception if this happens. This same consideration applies to all modes.

PUSH SP Instruction

The 8086 processor increments the content of the SP register before the value is pushed onto the stack, in contrast to the Super386 processor, which increments the content of the stack pointer after the value is pushed onto the stack. The 8086 PUSH SP instruction should be replaced with the following code:

```
PUSH BP
MOV BP, SP
XCHG BP, [BP]
```

PUSHF Instruction

On the 8086 processor, bits 15 through 12 of the FLAGS register are undefined. The PUSHF instruction on the 8086 processor sets these bits to 1. On the Super386 processor, bit 15 is cleared and bits 14 through 12 have the last values written to them.

Current Privilege Level

Because the 8086 does not support protection by privilege level, the following instructions that load descriptor tables cannot be executed in virtual-8086 mode:

- CLTS
- HLT
- LGDT
- LIDT
- LMSW
- MOV instructions that load or store the control registers.

These instructions cause a general-protection exception, which brings the processor to protected mode to emulate the instruction.

LOCK Prefix

The LOCK instruction prefix in a Super386 program will lock only the memory area designated by the destination operand. In 80286 and 8086 programs, LOCK causes the entire physical memory space to be locked. Therefore, use LOCK only with the following instructions, when the instruction writes to memory.

ADC	BTS	OR
ADD	DEC	SBB
AND	INC	SUB
BTC	NEG	XCHG
BTR	NOT	XOR

An invalid-opcode exception results from using LOCK with other instructions. See Appendix C for more details.

Bus Hold

The 8086 does not respond to requests for bus control, whereas the Super386 process will respond to inputs on its HOLD signal.

Interrupts and Exceptions

Some noteworthy interrupts and exceptions in virtual-8086 mode are discussed in the following paragraphs.

NMI Interrupts

When the Super386 processor is executing a nonmaskable interrupt, all other NMIs are masked until an IRET instruction is executed.

Instructions That Cross Offsets 0 or 65535

If an instruction accesses a memory operand beyond the 64kB maximum offset permitted on the 8086 processor, or if it accesses a memory operand with an offset of zero, the instruction wraps around the boundary and does not generate an exception. The Super386 processor generates a general-protection exception (or a stack exception, if a stack segment is addressed).

If a series of instructions pass the 64kB maximum offset, the 8086 processor retrieves the next byte of the instruction from the zero offset location. The Super386 processor generates a general-protection exception.

Single-Step Interrupt Priority

The 8086 external interrupt handler cannot be used for single-step operations when an interrupt occurs. The Super386 processor will single-step through an interrupt because the single-step interrupt has a higher priority than other interrupts.

Coprocessor Interrupt Controller

The coprocessor error signal, 8087 INT, passes through an interrupt handler. You may have to delete some instructions in a coprocessor handler if they operate with the interrupt controller.

DIV Instruction

For divide exceptions on the 8086 processor, the instruction pointer points to the next instruction; on the Super386 processor, it points to the instruction that failed. The 8086 generates a divide-error exception when IDIV generates a large negative number. The Super386 processor can operate with these large quotients.

Super386 Opcodes

Super386 opcodes generate an invalid-opcode exception if they are not defined in the 8086 code.

Coprocessor Errors

If the 8086 program does not use interrupt 16 for the coprocessor-error exception, the vector for both interrupt 16 and the one used by the 8086 program must point to the same coprocessor-error exception handler.

Executing Protected-Mode 80286 Code

Programs written for the protected-mode 80286 processor run on a Super386 processor without modification, because 80286 object code is a subset of Super386 object code. However, you may need to make some programmatic changes to ensure execution without exceptions. The differences between the two processors, presented in this section, affect operating systems more than application programs.

Task State Segments

All 16-bit 80286 TSSs should be changed to 32-bit Super386 TSSs without changing the object modules. This improves performance and allows paging to be used. We recommend that all TSSs be changed, because there are potential operating system problems if TSSs from both environments are used in the same program.

Paging

Paging can be used to map the first 64kB of address space beyond the 1MB limit of the address space to the lower part of the segment. This will compensate for the difference in wrap-around between the 80286 and Super386 processors. To use paging, however, the TSSs should first be modified to Super386 TSSs, as described in the section "Task State Segments".

The LOCK Prefix

The LOCK prefix should only be used to protect data operations from interruption by another bus master. In hardware system designs that observe the Super386 architecture, a locked instruction locks only the memory area designated by the destination operand. In the 80286 and 8086 architecture, the LOCK prefix causes the entire physical memory space to be locked. See Appendix C for more details.

Use LOCK only with the following instructions, when the instruction changes the contents of memory.

ADC	BTS	OR
ADD	DEC	SBB
AND	INC	SUB
BTC	NEG	XCHG
BTR	NOT	XOR

An invalid-opcode exception will result from using LOCK with other instructions.

Segment Descriptors

The Super386 processor supports all 80286 descriptors: code segments, data segments, local descriptor tables, task gates, TSSs, call gates, interrupt gates, and trap gates. For TSSs, call gates, interrupt gates, and trap gates, the Super386 supports its own 32-bit version of the descriptors as well as the 16-bit 80286 version. The default address/operand-size bit (D bit) in the code segment descriptors denotes whether the code segment should behave as an 80286 or Super386 segment. However, note the differences discussed in the following paragraphs.

Code Segment Descriptor—Set the default address/operand-size bit (D bit) to 1 for 32-bit operation. Clear the bit to 0 for 16-bit operation.

Stack Segment Descriptor—Set the big bit (B bit) to 1 to select the 32-bit ESP register. Clear it to 0 to select the 16-bit SP register.

Granularity—The G bit in all segment descriptors determines the maximum segment size. When cleared to 0, it specifies a byte-granular segment to a limit of 2^{20} bytes. When set to 1, it specifies a page-granular segment to a limit of 2^{32} bytes.

Base Address—In the 80286 format, the most-significant byte of the dword address is all zeros. In the Super386 format, all 32 bits of a dword can be used.

Limit Field—The most-significant four bits of the 20-bit segment limit are cleared to 0 in 80286 programs, which permits only a 64kB segment limit. In the Super386 format, segment limits can be 32 bits.

Type Field—There are differences between the type fields of segment descriptors on the 80286 processor and the Super386 processor. See the section entitled "Segment Descriptors" for details.

Reserved Word—In 80286 architecture, the most significant word of every 8-byte descriptor is reserved. In 80286 code, this word should be used to store zeros. Strange errors may occur if this upper word contains anything except zeros.

Mixing 32-bit and 16-bit Stacks

Do not use 16-bit gates. If a system call from privilege level 3 comes from a 32-bit stack frame through a 16-bit gate, the most significant 16 bits of the ESP stack pointer will be lost. To avoid this, all system calls should go through 32-bit gates.

Because interrupts can occur before execution of an instruction with the RF flag set, the RF flag may not function as expected when executing 16-bit code. The interrupt will occur in this case, pushing only 16 bits of the EFLAGS register on the stack. After the interrupt handler clears RF and returns to the original program, FLAGS will be restored but RF will not, because it is in the upper 16 bits. As a result, multiple instruction debug faults will occur.

IOPL Check

On the 80286 processor, I/O instructions and the LOCK instruction prefix are sensitive to the I/O privilege level (IOPL). A general-protection exception will be generated if the CPL is higher than the IOPL. On the Super386 processor, no checking is performed against the IOPL in real mode or virtual-8086 mode; such checks are only performed in protected mode.

Using Intermixed Word and Dword Operands

The Super386 processor is object-code compatible with the 8086, 8088, 80186, 80188, and 80286 and runs existing software written for these processors. This requires the ability to operate with 16-bit and 32-bit operands and addresses in the same program. However, the following principles and protocols of the Super386 processor must be observed for these programs to function properly.

Operand Size—Operand size is either byte or word/dword by default. To handle a non-default operand size, use instruction prefixes.

Pointers—Code and data pointers have either 16-bit or 32-bit offsets, depending on the setting of the D bit in the code segment descriptor.

Control Transfer—Control is transferred between 16-bit or 32-bit segments using call gates, trap gates, and interrupt gates. The operand size is determined by the type of gate.

Segment Limits—Segments can be up to 4GB, versus 64kB for a 16-bit environment.

The following sections describe the methods that enable the Super386 processor to operate with 16-bit and 32-bit operands and addresses.

Default Operand and Address Size

The default or D bit (bit 22) in code-segment descriptors sets the operand-size and address-size default for all operations related to the segment. Using the D bit saves an instruction prefix byte when all operands and addresses are one size. Setting D to 1 specifies 32-bit size; clearing D to 0 specifies 16-bit size. All 8086 programs have 16-bit attribute sizes. If a segment contains code of both sizes, 16-bit pointers can only access the first 64kB of the segment. Data segments with a limit equal to or less than 64kB can be shared by 32-bit and 16-bit pointers.

Stack Pointer Size

The big or B bit (bit 22) in a stack data-segment descriptor specifies the size of the stack pointer stored in the 32-bit ESP (or 16-bit SP) stack pointer register. When a dword is pushed onto the stack, the ESP register is decremented by 4; when a word is pushed onto the stack, the ESP register is decremented by 2.

The stack pointer size must be chosen carefully to accommodate these situations in mixed 16-bit and 32-bit code. Use the guidelines listed below to keep the stack pointer on word boundaries for 16-bit operation or on dword boundaries for 32-bit operation.

- When a segment register is pushed or popped, the operand size always matches the default size specified in the code segment (the D bit).
- When a 32-bit task state segment (TSS) is referenced, a dword is pushed onto the stack.
- When a 16-bit TSS is referenced, a word is pushed onto the stack.

Other B-bit Parameters

A data segment descriptor's B bit indicates the upper bound for a stack segment, the descriptor limit, and the point where wrap-around occurs when an access reaches the limit. The upper bound is FFFFFFFFh when B = 1, and FFFFh when B = 0.

Instruction Prefixes

The operand-size instruction prefix (66h) and the address-size instruction prefix (67h) toggle the instruction's default size, which is specified by the D bit in the code segment descriptor. Instruction prefixes can be used in any execution mode. The operand-size prefix is used where operands in a segment are of the non-default size. The address-size prefix is used to address an operand in a segment with a limit that exceeds 64kB when in 16-bit mode.

16-bit and 32-bit Registers

The D bit in the code-segment descriptor should be set to either dword or word size, depending on the size of the largest proportion of your code's operands and addresses. This saves an extra prefix byte in the instruction.

This choice, however, affects the way in which linear addresses are calculated. A modulus of 64kB is used to calculate 16-bit linear addresses, whereas 32-bit addresses do not use a modulus. The following comparison illustrates this difference:

```
16-bit address:    (index + base + displacement) MOD 64K + segment base
32-bit address:    index + base + displacement + segment base
```

Trap Gates and Interrupt Gates

When an operand passes through a trap gate or an interrupt gate, the gate size (32 or 16 bits) controls the resulting operand size.

APPENDIX A

The Super386 Instruction Set

This appendix contains an alphabetical list of all instructions and some prefixes available to the Super386 microprocessors. Appendix B contains quick reference tables covering exceptions and addressing modes. Refer to Appendix C for special programming considerations.

Notations

The following list identifies notations and abbreviations used in the tabulated instruction set throughout this appendix:

AF	Auxiliary flag
AH	Upper byte of the AX register
AL	Lower byte of the AX register
BH	Upper byte of the BX register
BL	Lower byte of the BX register
CF	Carry flag
CH	Upper byte of the CX register
CL	Lower byte of the CX register
cr	Control register
CS	Code segment register
DF	Direction flag
DH	Upper byte of the DX register
DL	Lower byte of the DX register
dr	Debug register
DS	Data segment register
dst	Destination operand, usually the first operand in the instruction.

(E)AX	The AX or EAX general register
(E)BP	The BP or EBP general register
(E)BX	The BX or EBX general register
(E)CX	The CX or ECX general register
(E)DI	The DI or EDI general register
[(E)DI]	Same as (E)DI, except that this is an address contained in the register.
(E)DX	The DX or EDX general register
(E)IP	The IP or EIP instruction pointer
ES	The ES segment register
(E)SI	The SI or ESI general register
[(E)SI]	Same as (E)SI, except that this is an address contained in the register.
(E)SP	The SP or ESP general register
FS	The FS segment register
GS	The GS segment register
IF	Interrupt flag
imm	A value encoded into the last field of the instruction that can be used directly
imm8	An imm value specified as byte-size
imm16	An imm value specified as word-size
m	A memory operand encoded in the r/m field of the MODr/m byte
[m]	Same as m, except that this is an address contained in a memory operand.
m16	A memory operand m specified as word-size
m32	A memory operand m specified as dword-size
m64	A memory operand m specified to be qword-size (quadword-size)
m80	A memory operand m contained in 10 bytes
moff	A word or dword offset for a data value; follows the opcode byte.
NT	Nested task flag
OF	Overflow flag
PF	Parity flag
r	A general register encoded in the reg field of the MODr/m byte
r8	A general register r specified as byte-size
r16	A general register r specified as word-size
r32	A general register r specified as dword-size

reg	A word or dword general register encoded in bits 2:0 of the opcode
rel	A word or dword field added to the (E)IP to calculate the address for a JMP or CALL
rel8	A rel value specified as byte-size
RF	Resume flag
r/m	A memory operand or general register, encoded in the r/m field of the MODr/m byte. The operand can be byte-size, determined by the opcode, or word/dword size depending on the operand size attribute.
[r/m]	Same as r/m, except that this is an address or offset contained in a memory operand or general register.
r/m8	An r/m value specified as byte-size
r/m16	An r/m value specified as word-size
r/m32	An r/m value specified as dword-size
sel	Segment selector
SF	Sign flag
src	The source operand, usually the second operand in the instruction
SS	Stack segment register
TF	Trap flag
tr	Test register
/x	Where x is replaced with a digit 0-7; indicates that the reg field of the MODr/m byte is used to further specify the opcode. For example, in MUL opcode F6 /4 the opcode byte is F6h and the reg field of the MODr/m byte is 4.
VF	Virtual-8086 mode bit in the EFLAGS register
ZF	Zero flag
{8}	Indicates opcode for byte size operands.
{16, 32}	Indicates opcode for operand whose size is determined by the default (D bit) and an included instruction prefix.

Register Encoding

Tables A-1 and A-2 give the encodings used for registers in the MODr/m byte.

Table A-1. *General Register Encoding*

Register Code	32-Bit Register	16-Bit Register	8-Bit Register
000b	EAX	AX	AL
001b	ECX	CX	CL
010b	EDX	DX	DL
011b	EBX	BX	BL
100b	ESP	SP	AH
101b	EBP	BP	CH
110b	ESI	SI	DH
111b	EDI	DI	BH

Table A-2. *Segment Register Encoding*

Register Code	Segment Register
000b	ES
001b	CS
010b	SS
011b	DS
100b	FS
101b	GS
110b	Reserved
111b	Reserved

Clock Counts

The clock parameters for each instruction indicate the number of clock cycles required to execute the instruction. These clock counts are based on the assumptions described below. Other inter-instruction events—including operand conflicts, operand alignment, and external bus wait or hold states—may increase the number of clock cycles required for execution. These events are also described below.

Notations

*	When an asterisk (*) follows a clock count (e.g., 2*), the instruction may require an additional cycle to decode. However, this additional clock is only needed when (a) the preceding instruction executes in one clock, or (b) the preceding instruction is a taken jump. Most instructions are decoded in a pipeline, so that this additional decode clock does not have an observable effect on execution speed.
/	When a slash (/) separates two clock counts (e.g., 11/13), the first count applies to the operand in a register and the second applies to the operand at a memory location.
n	When the letter n follows a clock count (e.g., 13+6n*), the count to which the n is appended must be multiplied by the number of iterations in the operation.
rm	Real mode
pm	Protected mode
vm	Virtual-8086 mode

Basic Assumptions

The assumptions behind the clock count values are the following:

- The external bus is available for reads or writes. If it is not, add clocks to reads and writes until the bus becomes available.
- Accesses are aligned. If they are not, add two clocks.
- Instruction-cache fills complete before subsequent cache accesses. If the instruction being requested by the instruction prefetcher is currently being written to the instruction cache, the cache will be bypassed. There is no operand cache, so operand accesses are not delayed in this manner.
- If an effective address is calculated, the base register is not the destination register of a preceding fetch. If this occurs, add two clocks.
- Effective address calculations do not use an index register. When they do, add one clock.
- The target of a jump is in the cache. If it is not, add four clocks. If the target is not completely contained in the first dword read, add two clocks.
- Writes are never delayed. There are no write buffers.
- If an instruction contains a displacement or immediate operand, the latter is contained within the first four bytes of the instruction. If it is not, add one clock.
- Operand accesses are not delayed by invalidate cycles. The instruction prefetcher may be delayed, but because the prefetch unit queues instructions, the effect is not often seen.
- Page translation hits in the TLB. If it does not, add 7, 12, or 17 clocks, depending on whether the accessed and/or dirty bit in neither, either, or both the page directory and page table need to be set.
- No exceptions are generated during instruction execution.

Operand Conflicts

Operand conflicts occur when an instruction's operand is being altered by a previous instruction. Due to the design of the instruction pipeline, these conflicts never occur for memory operands. Register operands, however, can cause such conflicts. In general, conflicts occur when an instruction moves an operand from memory into a register (either a load or a pop), and the following instruction requires the register and does not itself include a memory operand. If this occurs, add two clocks. The computation of clock delays for other types of operand conflict are more complex.

External Bus Wait and Hold States

If the bus needs more than two clocks to process a request, wait states are inserted. Alternatively, the bus may be unavailable. This can happen if instructions are being fetched or if an external device controls the bus. In all such cases, the instruction pipeline will wait until the operation can be completed. For store operations, the pipeline will only wait for the operation to be initiated. This allows instruction execution to continue even if the store needs many cycles.

Instruction Prefixes

The prefixes listed in Table A-3 can be used with instructions. If the prefixes can only be used under certain conditions, these conditions are described in the table and/or in the text of this appendix that describes specific instructions.

The address-size prefix is only meaningful for memory operands. Including multiple segment-select prefixes is of no value, as only one memory operand of any instruction can ever be overridden. The PUSH mem instruction has one memory operand that is fixed (pointed to by the stack pointer), and one that can be altered (the memory location pointed to as defined by the MODr/m byte). A segment select prefix alters the operand segment specified by the MODr/m byte.

Some instructions access multiple memory operands from a single address. BOUND reads two operands from memory, the first located at the address specified by the MODr/m byte, and the second located at an address either 2 or 4 bytes higher, depending on the operand size. If the default segment is overridden, both operands will come from the same newly selected segment.

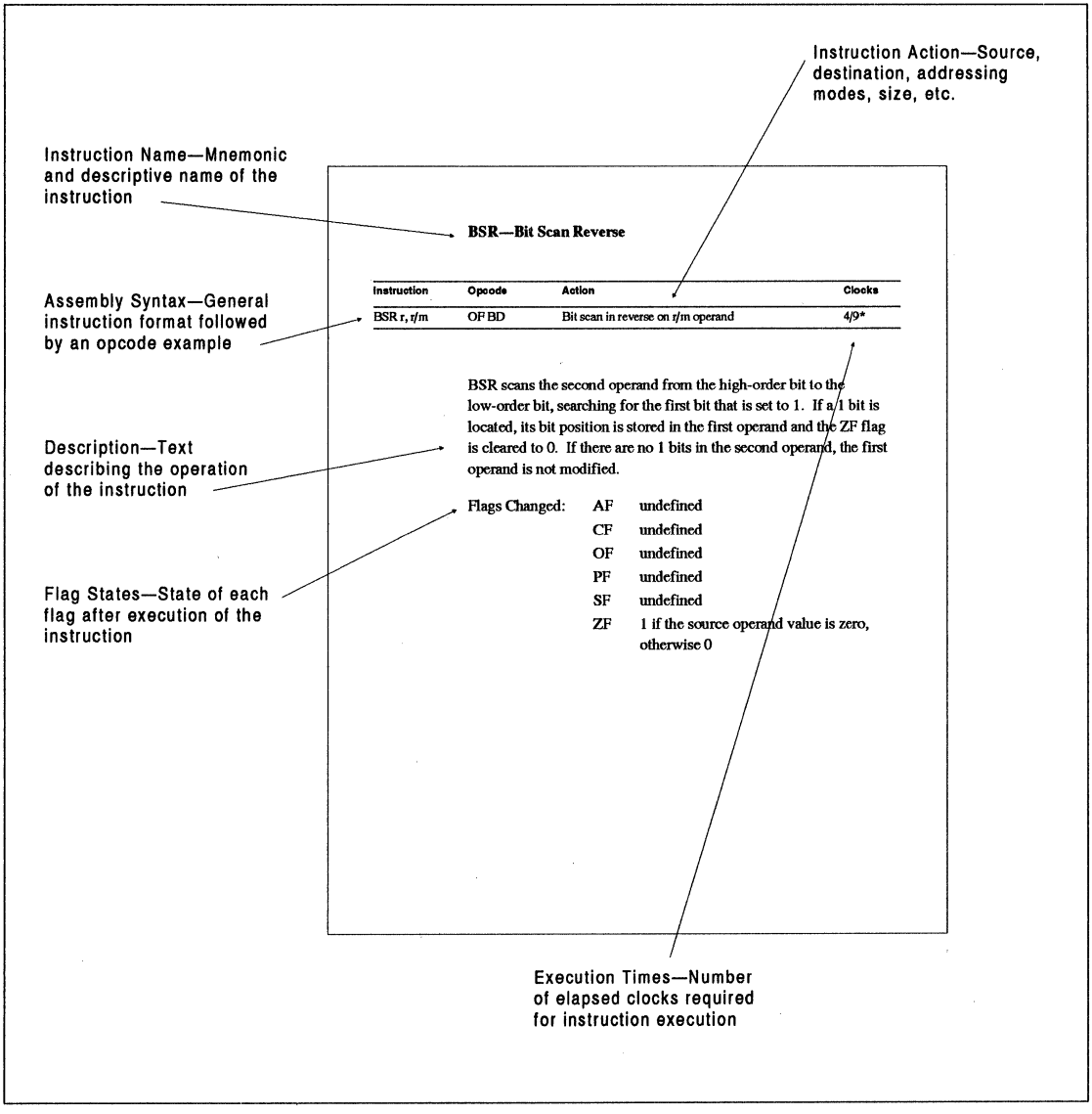
Table A-3. *Instruction Prefixes*

Type	Register or Prefix Name	Prefix Code	Description
Segment Override	CS	2Eh	Use CS segment for memory operand.
	DS	3Eh	Use DS segment for memory operand.
	ES	26h	Use ES segment for memory operand.
	FS	64h	Use FS segment for memory operand.
	GS	65h	Use GS segment for memory operand.
	SS	36h	Use SS segment for memory operand.
Operand Size		66h	Make operand-size attribute the opposite of the default (16-bit or 32-bit). This attribute specifies the width of operands, whether word or dword. The default is determined by the default size bit (bit 22) of the current code segment descriptor. The prefix is only used with instructions that access words or dwords; it is ignored if used on instructions that access bytes. In most assemblers, the prefix is provided automatically for instructions whose implied operand size does not match the default operand size for the segment in which the operand appears.
Address Size		67h	Make address-size attribute the opposite of the default (16-bit or 32-bit). This attribute specifies the width of the offset for instruction addresses. The default is determined by the default size bit (bit 22) of the current code segment descriptor. The prefix is only used with instructions that access memory; otherwise it is ignored. In most aasemblers, the prefix is provided automatically for instructions whose implied address size does not match the instruction's code-segment address size.
Lock	LOCK	F0h	Assert the bus lock signal between memory read and write.
Repeat	REP or REPE	F3h	Repeat following string instruction.
	REPNE	F2h	Repeat following string instruction.

Instruction Descriptions

The instructions are described in the following pages, which are arranged alphabetically according to the instruction mnemonic. Figure A-1 illustrates how the information is presented for each instruction.

Figure A-1. *Instruction Example*



AAA—ASCII-Adjust AL After ADD

Instruction	Opcode	Action	Clocks
AAA	37	Convert result of addition in AL to allow conversion to ASCII	3

AAA converts two unpacked BCD digits to a valid unpacked BCD result after an addition (ADD) operation. To convert the result of an AAA instruction to ASCII, execute the instruction OR AL, 30h.

AAA checks to see whether the lower nibble in the AL register is greater than 9, or whether the AF flag is set to 1. If either is true, (a) the result in AL is converted to the correct BCD result by adding 6 to the lower nibble and clearing the upper nibble, (b) the AH register is incremented by 1, and (c) the AF and CF flags are set to 1. If the lower nibble in AL is less than 9, the AF and CF flags are cleared to 0 and the AH register is not modified.

Flags Changed:

AF	0 if no decimal carry from low nibble, 1 if carry
CF	0 if no decimal carry from low nibble, 1 if carry
OF	undefined
PF	undefined
SF	undefined
ZF	undefined

AAD—ASCII-Adjust AX Before Divide

Instruction	Opcode	Action	Clocks
AAD	D5 0A	Convert contents of AX before division to allow an ASCII result	9*

AAD is used before an unsigned integer division (DIV) to convert two unpacked BCD digits in the AX register to a valid unpacked BCD result in the AX register. After the divide operation, the result can be converted to ASCII representation by execution of the instruction OR AL, 30h.

The instruction assumes that the dividend is represented in the AX register by two BCD digits, the upper digit in AH and the lower digit in AL. The dividend is converted to its binary equivalent by placing the value AL + (10 * AH) in the AL register and clearing AH to zero.

Flags Changed:

AF	undefined
CF	undefined
OF	undefined
PF	0 if odd parity, 1 if even parity
SF	AL bit 7
ZF	0 if result was nonzero, 1 if result was zero

AAM—ASCII-Adjust AX After Multiplication

Instruction	Opcode	Action	Clocks
AAM	D4 0A	Converts result of multiplication in AX to allow conversion to ASCII	15*

AAM converts two unpacked BCD digits in the AX register to a valid unpacked BCD result in the AX register after an unsigned integer multiplication (MUL). To subsequently convert the result of an AAM instruction to ASCII, execute the instruction OR AL, 30h.

The product of the multiplication is assumed to be between 0 and 81 and is therefore contained entirely within the low-order byte of the AX register (the AL register). The AAM instruction converts this product from a binary value into two unpacked BCD digits by dividing the value in AL by 10, storing the resulting high-order BCD digit (quotient) in AH, and storing the resulting low-order BCD digit (remainder) in AL.

Flags Changed:

AF	undefined
CF	undefined
OF	undefined
PF	0 if odd parity, 1 if even parity
SF	AL bit 7
ZF	0 if result was nonzero, 1 if result was zero

AAS—ASCII-Adjust AL After Subtract

Instruction	Opcode	Action	Clocks
AAS	3F	Alter results of subtraction in AL to allow conversion back to ASCII	3

AAS converts two unpacked BCD digits to a valid unpacked BCD result after a byte subtraction (the SUB, SBB, or NEG instructions only). To convert the result of an AAS instruction to ASCII, execute the instruction OR AL, 30h.

The instruction checks to see whether the lower nibble in the AL register is greater than 9 or whether the AF flag is set to 1. If either is true, (a) the result in AL is converted to the correct BCD result by subtracting 6 from the lower nibble and clearing the upper nibble, (b) the AH register is decremented by 1, and (c) the AF and CF flags are set to 1. If the lower nibble in AL is less than 9, the AF and CF flags are cleared to 0 and the AH and AL registers are not modified.

Flags Changed:

AF	0 if no decimal carry from low nibble, 1 if carry
CF	0 if no decimal carry from low nibble, 1 if carry
OF	undefined
PF	undefined
SF	undefined
ZF	undefined

ADC—Signed and Unsigned Integer Addition, With Carry

Instruction	Opcode	Action	Clocks
ADC r, r/m	12 {8}, 13 {16,32}	Add operand from r/m to r; add carry	1/5
ADC r/m, r	10 {8}, 11 {16, 32}	Add operand from r to r/m; add carry	1/5
ADC r/m, imm	80 /2 {8}, 81 /2 {16, 32}	Add imm operand to same-size r/m; add carry	1/5
ADC r/m, imm8	83 /2 {16, 32}	Add imm8 operand to r/m; add carry	1/5
ADC AL, imm8	14	Add imm8 operand to AL; add carry	1
ADC (E)AX, imm	15 {16, 32}	Add imm operand to (E)AX; add carry	1

ADC adds the first operand, second operand, and carry flag and then stores the result in the first operand. When an immediate byte is added to a word or dword operand, it is sign-extended to the size of the operand.

The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:

AF	0 if no carry from low nibble, 1 if carry
CF	0 if no carry from high-order bit, 1 if carry
OF	0 if no overflow, 1 if overflow
PF	0 if odd parity, 1 if even parity
SF	high-order bit of result
ZF	0 if result was nonzero, 1 if result was zero

ADD—Signed and Unsigned Integer Addition

Instruction	Opcode	Action	Clocks
ADD r, r/m	02 {8}, 03 {16, 32}	Add operand from r/m to r	1/5
ADD r/m, r	00 {8}, 01 {16, 32}	Add operand from r to r/m	1/5
ADD r/m, imm	80 /0 {8}, 81 /0 {16, 32}	Add imm operand to same-size r/m	1/5
ADD r/m, imm8	83 /0 {16, 32}	Add imm8 operand to r/m	1/5
ADD AL, imm8	04	Add imm8 operand to AL	1
ADD (E)AX, imm	05 {16, 32}	Add imm operand to (E)AX	1

The ADD instruction adds the first and second operands and then stores the result in the first operand. Before an immediate byte is added to a word or dword operand, the byte is sign-extended to the size of the word or dword operand.

If ADD is used on packed BCD digits, the DAA instruction can be used subsequently for decimal adjustment. If ADD is used on unpacked BCD digits, the AAA instruction can be used subsequently for adjustment prior to ASCII conversion.

The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:

AF	0 if no carry from low nibble, 1 if carry
CF	0 if no carry from high-order bit, 1 if carry
OF	0 if no overflow, 1 if overflow
PF	0 if odd parity, 1 if even parity
SF	high-order bit of result
ZF	0 if result was nonzero, 1 if result was zero

AND—Bitwise Logical AND

Instruction	Opcode	Action	Clocks
AND r, r/m	22 {8}, 23 {16, 32}	Logical AND of r/m and r operands, result in r	1/5
AND r/m, r	20 {8}, 21 {16, 32}	Logical AND of r/m and r operands, result in r/m	1/5
AND r/m, imm	80 /4 {8}, 81 /4 {16, 32}	Logical AND of r/m and imm operands, result in r/m	1/5
AND r/m, imm8	83 /4 {16, 32}	Logical AND of r/m and imm8 operands, result in r/m	1/5
AND AL, imm8	24	Logical AND of AL and imm8 operands, result in AL	1
AND (E)AX, imm	25 {16, 32}	Logical AND of (E)AX and imm operands, result in (E)AX	1

The AND instruction performs a logical AND on the two operands. The result is stored in the destination operand.

In AND operations, a 1 bit is written when both corresponding bits in the operands are 1, otherwise 0 is written. The instruction is useful for masking (clearing to 0) specific bits in a number. For example, ANDing the binary value 0111 1111 with any number will clear its most-significant (sign) bit.

The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:

AF	undefined
CF	0
OF	0
PF	0 if odd parity, 1 if even parity
SF	high-order bit of result
ZF	0 if result was nonzero, 1 if result was zero

ARPL—Adjust RPL Field of Selector

Instruction	Opcode	Action	Clocks
ARPL r/m16, r16	63	If RPL of r/m16 is less than RPL of r16, set RPL of r/m16 equal to RPL of r16	11/13*

ARPL compares the RPL field—the two low-order bits—of the two segment selectors contained in the operands. If the first (destination) selector has a lower numeric RPL (more privilege) than the second (source) selector, the source selector RPL overwrites the destination selector's RPL, and the ZF flag is set to 1.

The instruction is typically used in operating system procedures to ensure that a far call by an application program does not pass a segment selector having an RPL of greater privilege than the calling application's CPL. The operating system procedure can use the ARPL instruction by loading the selector to be passed in the destination operand and the caller's code segment selector (which contains the caller's CPL in its RPL field) in the source operand.

By using the ARPL instruction in this manner for each call procedure, the operating system can ensure that the RPL of a selector passed by an application program will not gain privilege if it is passed through a chain of procedures, some with higher CPL than others. The ARPL instruction ensures this by applying the following checking rule at each step of the chain:

$$RPL = Max(RPL_{caller}, CPL_{caller})$$

For details on privilege-level checking, see the sections entitled "Protection Mechanisms" and "Other Processing Modes" in Chapter 4.

Flags Changed: ZF = 1 if first-operand RPL is less than second-operand RPL, otherwise 0

BOUND—Check Array Index Against Bounds

Instruction	Opcode	Action	Clocks
BOUND r, m	62	Determine whether r is within bounds	15*

BOUND tests the first operand against upper and lower boundary values stored in the second operand.

The first operand, stored in a register, is a signed array index. The second operand, a data structure in memory, contains the high and low boundary values of the array. The two boundary values occupy two consecutive locations in memory—one word apart for 16-bit operands or one dword apart for 32-bit operands. The second operand points to the low boundary value in memory, and the next word or dword is the high value.

If the operation determines that the array index is out of bounds, a bound-range fault (exception 5) is generated.

Flags Changed: None

BSF—Bit Scan Forward

Instruction	Opcode	Action	Clocks
BSF r, r/m	0F BC	Bit scan forward on r/m operand	4/9*

BSF scans the second operand from the low-order bit to the high-order bit, searching for the first bit that is set to 1. If a 1 bit is located, its bit position is stored in the first operand and the ZF flag is cleared to 0. If there are no 1 bits in the second operand, the first operand is not modified.

Flags Changed:

AF	undefined
CF	undefined
OF	undefined
PF	undefined
SF	undefined
ZF	1 if the source operand value is zero, otherwise 0

BSR—Bit Scan Reverse

Instruction	Opcode	Action	Clocks
BSR r, r/m	0F BD	Bit scan in reverse on r/m operand	4/9*

BSR scans the second operand from the high-order bit to the low-order bit, searching for the first bit that is set to 1. If a 1 bit is located, its bit position is stored in the first operand and the ZF flag is cleared to 0. If there are no 1 bits in the second operand, the first operand is not modified.

Flags Changed:

AF	undefined
CF	undefined
OF	undefined
PF	undefined
SF	undefined
ZF	1 if the source operand value is zero, otherwise 0

BT—Bit Test

Instruction	Opcode	Action	Clocks
BT r/m, r	0F A3	Copy bit r of operand r/m into CF	2/8*
BT r/m, imm8	0F BA /4	Copy bit imm8 of operand r/m into CF	2/6*

BT reads the bit in the first operand at the position specified by the second operand, and assigns the bit's value to the CF flag.

If the first operand is a register, the bit offset is specified by the value in the second operand, module 16 or 32 (the destination's register size). That is, only the lower four bits (for 16-bit registers) or five bits (for 32-bit registers) of the second operand are used as the binary pointer in the first operand.

If the first operand is a memory location, the effect depends on whether the second operand is an 8-bit immediate value or a register, as discussed in the following paragraphs.

If the second operand is an 8-bit immediate value, bit offsetting works as described above, except that the modulus of 16 or 32 is determined by the destination operand's memory size rather than a register size.

If the second operand is a register, the memory is treated as a bitmap whose base address is given by the first operand. The second operand provides a bit offset of 0 to 64kB for 16-bit registers, or 0 to 4GB for 32-bit registers. (The size of the memory bitmap is, of course, also limited by the size of the data segment in which it resides.) The processor will then determine which word or dword in memory contains the bit to be tested, and the processor will read only that word or dword before testing the bit.

The LOCK prefix cannot be used with this instruction.

Flags Changed:

AF	undefined
CF	bit in selected position of destination operand
OF	undefined
PF	undefined
SF	undefined
ZF	undefined

BTC—Bit Test and Complement

Instruction	Opcode	Action	Clocks
BTC r/m, r	0F BB	Copy bit r of operand r/m into CF; complement bit r	4/10*
BTC r/m, imm8	0F BA /7	Copy bit imm8 of operand r/m into CF; complement bit imm8	4/8*

BTC reads the bit in the first operand at the position specified by the second operand, assigns the bit's value to the CF flag, and complements the bit in the first operand.

See the description of the BT instruction for details on how the first and second operands are interpreted.

The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:

AF	undefined
CF	bit in selected position of destination operand
OF	undefined
PF	undefined
SF	undefined
ZF	undefined

BTR—Bit Test and Reset

Instruction	Opcode	Action	Clocks
BTR r/m, r	0F B3	Copy bit r of operand r/m into CF; clear bit r	4/10*
BTR r/m, imm8	0F BA /6	Copy bit imm8 of operand r/m into CF; clear bit imm8	4/8*

BTR reads the bit in the first operand at the position specified by the second operand, assigns the bit's value to the CF flag, and clears to 0 the bit in the first operand.

See the description of the BT instruction for details on how the first and second operands are interpreted.

The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:

AF	undefined
CF	bit in selected position of destination operand
OF	undefined
PF	undefined
SF	undefined
ZF	undefined

BTS—Bit Test and Set

Instruction	Opcode	Action	Clocks
BTS r/m, r	0F AB	Copy bit r of operand r/m into CF; set bit r	4/10*
BTS r/m, imm8	0F BA /5	Copy bit imm8 of operand r/m into CF; set bit imm8	4/8*

BTS reads the bit in the first operand at the position specified by the second operand, assigns the bit's value to the CF flag, and sets to 1 the bit in the first operand.

See the description of the BT instruction for details on how the first and second operands are interpreted.

The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:

AF	undefined
CF	bit in selected position of destination operand
OF	undefined
PF	undefined
SF	undefined
ZF	undefined

CALL (near)—Call Subroutine in Same Segment

Instruction	Opcode	Action	Clocks
CALL rel	E8	Call near procedure at offset rel from next instruction	7
CALL [r/m]	FF /2	Call near procedure at address in [r/m]	9/11

A near CALL branches to a location within the current code segment. The branch destination is specified by the operand. The branch is either direct (the operand is an offset from the current instruction pointer) or indirect (the operand is a register or memory location that contains the branch address). The instruction increments the instruction pointer, pushes it onto the stack, and transfers control to the branch location specified in the operand.

In the instruction's direct form, the branch destination is obtained by adding a signed offset to the address of the next instruction after the CALL instruction. The offset is stored in the 32-bit EIP register. If the operand size is 16 bits, the high word of EIP is cleared to 0.

In the instruction's indirect form, the branch destination is to a specified address within the current code segment.

Use the far CALL instruction for branching to locations in different code segments. Use the task CALL instruction to transfer control through a task gate or directly to a different task state segment. Call instructions, like jump instructions, clear the instruction pipeline.

Flags Changed: None

CALL (far)—Call Subroutine in Different Segment

Instruction	Opcode	Action	Clocks (by mode):	rm, vm	pm
CALL sel:off	9A	Call far procedure at address sel:off		19	See Table A-4
CALL [m]	FF /3	Call far procedure at address in [m]		21	See Table A-4

A far CALL branches to a location in a code segment different than the current code segment. The operand specifies a far pointer—either 48 bits or 32 bits, depending on the operand-size attribute. The pointing is either direct (the pointer is the operand) or indirect (the pointer is contained in a memory location).

In protected mode, the number of clock cycles required for the instruction depends on the destination of the call, as shown in Table A-4.

Table A-4. *Far CALL Clocks*

	CALL sel:off	CALL [m]
To a code segment	67	69
To a gate at same privilege level	69	71
To a gate at inner (more privileged) level	189	191

In real mode and virtual-8086 mode, the instruction increments the instruction pointer, pushes the CS and (E)IP values onto the stack, loads CS with the far pointer's selector, sets the CS descriptor base register to selector *16, and loads the (E)IP with the offset. For 16-bit operands, the upper word of EIP is cleared to 0.

In protected mode, the instruction increments the instruction pointer, pushes the CS:(E)IP value onto the stack, and uses the segment selector as an offset into a descriptor table. The descriptor to which the segment selector points may directly specify a code segment, or it may specify a call gate. When the selector specifies a call gate, the call gate selector and offset are used for the address, and the offset value in the instruction is ignored. Call gates are described in the section entitled "Control Gates and System Calls" in Chapter 4.

For details on privilege-level checking, see the sections entitled "Protection Mechanisms" and "Other Processing Modes" in Chapter 4.

Near calls execute faster than far calls when branching to locations within the current code segment. See the description of the task CALL, which is the same opcode as the far CALL. Call instructions, like jump instructions, clear the instruction pipeline.

Flags Changed: None

CALL (task)—Call Different Task (Switch Task)

Instruction	Opcode	Action	Clocks
CALL sel:off	9A	Call task at address sel:off	See Table A-5
CALL [m]	FF /3	Call task at address in [m]	See Table A-5

A task CALL instruction works like a protected-mode far CALL, except that the task CALL selector specifies a TSS descriptor or a task gate descriptor, which in turn specifies a TSS descriptor. See the description of far CALL.

The number of clocks required for the instruction, in both of its forms and for calls to either a TSS descriptor or a task gate descriptor, depends on the type of source and destination task, as shown in Table A-5.

Table A-5. *Task CALL Clocks*

From	To Super386 Task	To 80286 Task	To Virtual-8086 Task
Super386 task	426	365	467
80286 task	419	358	460

If the new TSS descriptor is not busy, the current task state is saved in the existing TSS, and the new task state is loaded. The segment selector of the old TSS is saved in the back-link field of the new TSS, and the nested task (NT) flag is set to 1 in the new TSS. The section entitled "Multitasking" in Chapter 4 describes this mechanism, task gates, and TSSs.

Call instructions, like jump instructions, clear the instruction pipeline.

Flags Changed: All

CBW—Convert Byte to Word

Instruction	Opcode	Action	Clocks
CBW	98	Extend sign of AL through AX	2

CBW sign-extends the byte in the AL register to word length and places the result in the AX register. The value of the sign bit (bit 7) in the AL register is used to fill all bit positions of the AH register.

See the CDQ, CWD, and CWDE instructions.

Flags Changed: None

CDQ—Convert Doubleword to Quadword

Instruction	Opcode	Action	Clocks
CDQ	99	Extend sign of EAX through register pair EDX:EAX	2

CDQ sign-extends the dword in the EAX register to qword length and places the result in the EDX:EAX register pair. The value of the sign bit (bit 31) in the EAX register is used to fill all bit positions of the EDX register.

See the CBW, CWD, and CWDE instructions.

Flags Changed: None

CLC—Clear Carry Flag

Instruction	Opcode	Action	Clocks
CLC	F8	Clear CF to 0	2

CLC clears the carry flag (CF) to 0.

Flags Changed: CF = 0

CLD—Clear Direction Flag

Instruction	Opcode	Action	Clocks
CLD	FC	Clear DF to 0	2

CLD clears the direction flag (DF) to 0.

Following a CLD instruction, string instructions increment their index registers, (E)SI and/or (E)DI. The DF settings are:

DF = 1 Decrement (E)SI and (E)DI
DF = 0 Increment (E)SI and (E)DI

Flags Changed: DF = 0

CLI—Clear Interrupt Flag

Instruction	Opcode	Action	Clocks
CLI	FA	Clear IF to 0	3

CLI clears the interrupt flag (IF) to 0. When CLI is executed, the processor will not respond to external interrupt requests on the INTR signal until the IF flag is set to 1. Software interrupts (the various INT instructions) and the NMI hardware signal are not affected by the setting of this flag.

In protected mode and virtual-8086 mode, the CPL must be less than or equal to IOPL.

The flag is set with the STI instruction.

Flags Changed: IF = 0

CLTS—Clear Task-Switched Flag in CR0

Instruction	Opcode	Action	Clocks
CLTS	0F 06	Clear TS to 0	10*

CLTS clears bit 3 of the CR0 register, the task-switched (TS) flag.

The processor sets TS to 1 during each task switch. It can be tested to monitor coprocessor activity. A coprocessor-not-available fault (exception 7) is generated if a coprocessor ESC instruction is executed while the TS flag is set to 1, or if a WAIT instruction is executed with the MP and TS flags both set to 1. See the sections entitled "System Registers" and "Multitasking" in Chapter 4.

The instruction can only be executed at privilege level 0.

Flags Changed: TS (in CR0) = 0

CMC—Complement Carry Flag

Instruction	Opcode	Action	Clocks
CMC	F5	Complement CF	2

CMC toggles the carry flag (CF). If it was set to 1, CMC clears it to 0, and vice versa.

For explicit setting of the flag, use the CLC or STC instructions.

Flags Changed: CF = complement of CF

CMP—Compare Operands

Instruction	Opcode	Action	Clocks
CMP r, r/m	3A {8}, 3B {16, 32}	Compare r/m and r	1/5
CMP r/m, r	38 {8}, 39 {16, 32}	Compare r/m and r	1/5
CMP r/m, imm	80 /7 {8}, 81 /7 {16, 32}	Compare r/m and imm	1/5
CMP r/m, imm8	83 /7 {16, 32}	Compare r/m and imm8	1/5
CMP AL, imm8	3C	Compare AL and imm8	1
CMP (E)AX, imm	3D {16, 32}	Compare (E)AX and imm	1

CMP subtracts the second operand from the first operand. The arithmetic flags are set according to the result, but the result is not retained. If the operands are different sizes, the shorter operand is sign-extended before the subtraction.

The instruction is commonly used before a Jcc or SETcc instruction.

Flags Changed:

AF	0 if no borrow to low nibble, 1 if borrow
CF	0 if no borrow to high-order bit, 1 if borrow
OF	0 if no overflow, 1 if overflow
PF	0 if odd parity, 1 if even parity
SF	high-order bit of result
ZF	0 if result was nonzero, 1 if result was zero

CMPSB, CMPSW, and CMPSD—Compare Strings

Instruction	Opcode	Action	Clocks
CMPSB	A6	Compare byte at address in ES:[(E)DI] to byte at address in DS:[(E)SI]	9
CMPSW	A7	Compare word at address in ES:[(E)DI] to word at address in DS:[(E)SI]	9
CMPSD	A7	Compare dword at address in ES:[(E)DI] to dword at address in DS:[(E)SI]	9

These instructions subtract two strings in memory that are indirectly addressed by the contents of the ES:(E)DI and DS:(E)SI registers. The flags are set according to the result of the subtraction. The subtraction result itself is discarded.

The first operand, found at the address contained in ES:(E)DI, is subtracted from the second operand, found at the address contained in DS:(E)SI. This is opposite to the normal destination-source convention used, for example, in the SUB instruction.

The default segment for the second operand, DS, can be overridden with an instruction prefix, but the default for the first operand, ES, cannot.

If the DF flag is cleared to 0, the memory addresses contained in both the source and destination registers are incremented by 1, 2, or 4 (depending on operand size) to point to the next string element. If DF is set to 1, the registers are decremented. The LOOP instruction or the REP instruction prefix can be used to repeat the operation.

See the SCASB, SCASW, and SCASD instructions.

Flags Changed:

AF 0 if no borrow to low nibble, 1 if borrow
CF 0 if no borrow to high-order bit, 1 if borrow
OF 0 if no overflow, 1 if overflow
PF 0 if odd parity, 1 if even parity
SF high-order bit of result
ZF 0 if result was nonzero, 1 if result was zero

CWD—Convert Word to Dword

Instruction	Opcode	Action	Clocks
CWD	99	Extend sign of AX through register pair DX:AX	2

CWD sign-extends the word in the AX register to dword length and places the result in the DX:AX register pair. The high-order bit (bit 15) in the AX register is used to fill all bit positions of the DX register.

See the CBW, CDQ, and CWDE instructions. CWD is the word operand version of CDQ. CWDE performs the same sign extension as CWD, but it puts the results in EAX instead of DX:AX.

Flags Changed: None

CWDE—Convert Word to Dword Extended

Instruction	Opcode	Action	Clocks
CWDE	98	Extend sign of AX through register EAX	2

CWDE sign-extends the word in the AX register to dword length and places the result in the EAX register. The high-order bit (bit 15) in the AX register is used to fill all bit positions of the upper word of the EAX register.

See the CBW, CDQ, and CWDE instructions. CBW is the byte version of CWDE. CWD performs the same sign extension as CWDE, but it puts the results in DX:AX instead of EAX.

Flags Changed: None

DAA—Decimal Adjust AL After ADD

Instruction	Opcode	Action	Clocks
DAA	27	Convert packed BCD in AL to packed decimal after addition	3

DAA converts the result of binary addition on packed BCD digits to a valid decimal result. The instruction is used after an ADD or ADC instruction adds two packed BCD numbers and places the result in the AL register.

If the low-order nibble in AL is greater than 9, or the AF flag is set to 1, DAA adds 6 to the low-order nibble and sets the AF flag to 1. If AL is greater than 99h or the CF flag is set to 1, DAA adds 60h to AL and sets the CF flag to 1.

Flags Changed:

AF	1 when AL bits 3:0 are greater than 9
CF	1 when AL bits 7:4 are greater than 9
OF	undefined
SF	AL bit 7
PF	0 if odd parity, 1 if even parity
ZF	0 if result was nonzero, 1 if result was zero

DAS—Decimal Adjust AL After Subtract

Instruction	Opcode	Action	Clocks
DAS	2F	Convert packed BCD in AL to packed decimal after subtraction	3

DAS converts the result of binary subtraction on packed BCD digits to a valid decimal result. The instruction is used after a SUB or SBB instruction subtracts two packed BCD numbers and places the result in the AL register.

If the low-order nibble in AL is greater than 9 or the AF flag is set to 1, DAS subtracts 6 from the low-order nibble and sets the AF flag to 1. If AL is greater than 99h or the CF flag is set to 1, DAS subtracts 60h from AL and sets the CF flag to 1.

Flags Changed:

AF	1 when AL bits 3:0 are less than 0
CF	1 when AL bits 7:4 are less than 0
OF	undefined
SF	AL bit 7
PF	0 if odd parity, 1 if even parity
ZF	0 if result was nonzero, 1 if result was zero

DEC—Decrement by One

Instruction	Opcode	Action	Clocks
DEC r/m	FE/1 {8}, FF/1 {16,32}	Decrement r/m by 1	1/5
DEC reg	48+reg {16,32}	Decrement reg by 1	1

DEC decrements the destination operand by 1.

Unlike decrements performed by the SUB instruction, DEC does not modify the CF flag. The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:	AF	0 if low nibble is nonzero, 1 if low nibble is zero
	OF	0 if no overflow, 1 if overflow
	PF	0 if odd parity, 1 if even parity
	SF	high-order bit of result
	ZF	0 if result was nonzero, 1 if result was zero

DIV—Unsigned Divide

Instruction	Opcode	Action	Clocks
DIV r/m8	F6 /6	Divide AX by r/m8; quotient in AL, remainder in AH	15
DIV r/m16	F7 /6	Divide DX:AX by r/m16; quotient in AX, remainder in DX	23
DIV r/m32	F7 /6	Divide EDX:EAX by r/m32; quotient in EAX, remainder in EDX	39

DIV divides an unsigned dividend by an unsigned divisor and stores the resulting quotient and remainder. The locations of the elements, organized by size of the instruction operand, are shown in Table A-6. For dividends, the DX and EDX registers store the most significant bits.

Table A-6. *DIV Element Storage Locations*

Element	Byte	Word	Dword
Dividend	AX	DX:AX	EDX:EAX
Divisor	operand	operand	operand
Quotient	AL	AX	EAX
Reminder	AH	DX	EDX

Exception 0 is generated if a divide-by-zero fault occurs, or if the quotient does not fit in the quotient register. When a divide error occurs, the return address points to the divide instruction on entry to the exception handler.

See the IDIV instruction.

Flags Changed:

OF	undefined
SF	undefined
ZF	undefined
AF	undefined
PF	undefined
CF	undefined

ENTER—Create a Nested Stack Frame

Instruction	Opcode	Action	Clocks
ENTER imm16, 0	C8 {16, 32}	Make stack frame of size imm16	12
ENTER imm16, 1	C8 {16, 32}	Make stack frame of size imm16 at level 1	13
ENTER imm16, imm8	C8 {16, 32}	Make stack frame of size imm16 at level imm8	13+6n

The ENTER instruction creates a stack frame for a recursively callable procedure. It allocates the stack frame, creates pointers to the stack frames of procedures in which the new procedure is nested (called the display), and inserts a dynamic link to the calling procedure.

The first operand specifies the number of bytes needed for the procedure's local variables, not including the procedure's display or dynamic link. The second operand specifies the nesting level of the routine, from 0 (outermost) to 31 (innermost). The nesting level is the number of stack frame pointers in the display, including all pointers copied from the current stack frame to the new stack frame, plus one (which points to the newly created stack frame itself).

When the new stack frame is set up, the caller's (E)BP is pushed onto the current stack and the (E)BP register is updated to point to the new stack. The first operand is then subtracted from (E)SP.

The ENTER instruction is used at the beginning of a procedure. A LEAVE instruction is used to undo the effect of ENTER just prior to a return instruction.

Flags Changed: None

ESC—Escape to Coprocessor

Instruction	Opcode	Action	Clocks
ESC ecode, r	D8+fop/ext	Transfer instruction execution to coprocessor	See Coprocessor Document
ESC ecode, m	D8+fop/ext	Transfer instruction execution to coprocessor	See Coprocessor Document

ESC is a special prefix for a floating-point instruction. It causes the processor to pass the floating-point instruction to the coprocessor.

The number of clocks required for the instruction depends on the particular coprocessor being used (see documentation for the coprocessor). A coprocessor-not-available fault (exception 7) is generated if the code is encountered when no coprocessor is present.

See the WAIT instruction.

Flags Changed: None

HLT—Halt Processor

Instruction	Opcode	Action	Clocks
HLT	F4	Halt execution of instructions; restart on interrupt	4

HLT idles the processor, preventing it from executing instructions until the processor receives an NMI, enabled interrupt, or reset. The address to which control returns from an interrupt handler is contained in the CS:(E)IP register. It points to the instruction following the HLT instruction.

The instruction can only be executed at privilege level 0. HLT operates in real mode because the CPL is always 0, but it will generate a fault in virtual-8086 mode, which operates at privilege level 3.

Flags Changed: None

IDIV—Signed Divide

Instruction	Opcode	Action	Clocks
IDIV r/m8	F6 /7	Divide AX by r/m8; result in AL, remainder in AH	16
IDIV r/m16	F7 /7	Divide DX:AX by r/m16; result in AX, remainder in DX	24
IDIV r/m32	F7 /7	Divide EDX:EAX by r/m32; result in EAX, remainder in EDX	40

IDIV divides a signed dividend by a signed divisor and stores the resulting quotient and remainder. The location of the elements, organized by size of the instruction operand, are shown in Table A-7. For dividends, the DX and EDX registers store the most significant bits. The remainder has the same sign as the dividend. Quotients which are nonintegral are truncated toward 0.

Table A-7. *IDIV Element Storage Locations*

Element	Byte	Word	Dword
Dividend	AX	DX:AX	EDX:EAX
Divisor	r/m8	r/m16	r/m32
Quotient	AL	AX	EAX
Reminder	AH	DX	EDX

Exception 0 is generated if a divide-by-zero fault occurs, or if the quotient does not fit in the quotient register. When a divide error occurs, the return address points to the divide instruction on entry to the exception handler.

See the DIV instruction.

Flags Changed:

AF	undefined
CF	undefined
OF	undefined
PF	undefined
SF	undefined
ZF	undefined

IMUL—Signed Multiply

Instruction	Opcode	Action	Clocks
IMUL AL, r/m8	F6 /5	Multiply AL by r/m8; result in AX	8 to 12
IMUL AX, r/m16	F7 /5	Multiply AX by r/m16; result in DX:AX	8 to 16
IMUL EAX, r/m32	F7 /5	Multiply EAX by r/m32; result in EDX:EAX	8 to 24
IMUL r, r/m	0F AF {16, 32}	Multiply r by r/m; result in r	7 to 23
IMUL r, r/m, imm	69 {16, 32}	Multiply r/m by imm; result in r	9 to 24
IMUL r, r/m, imm8	6B {16, 32}	Multiply r/m by imm8; result in r	9 to 12

IMUL performs a signed multiplication and stores the result in a register or register pair.

In the first three forms of the instruction (implied accumulator, two operands), the two operands are multiplied and the result is stored in the registers AX, DX:AX, and EDX:EAX, respectively. The CF and OF flags are cleared to 0 when the multiplication produces the same result as would have been produced by sign-extending the multiplicand in AL/AX/EAX.

In the fourth form (explicit accumulator, two operands) the two operands are multiplied and the result is stored in the first operand. The CF and OF flags are cleared to 0 when the result fits exactly in the first register.

In the last two forms (explicit accumulator, three operands) the second and third operands are multiplied and the result is stored in the first operand. The CF and OF flags are cleared to 0 when the result fits exactly in the first register.

Before starting the multiplication operation, the processor determines which bits in the multiplier are significant to the value of the multiplier. The processor then performs the multiplication by examining, summing, and shifting two bits at a time in the multiplicand and multiplier, until all significant bits of the multiplier have been operated on. This is referred to as an early-out algorithm, because it allows the multiplication to terminate before all bits of the operands have been summed and shifted.

Clock counts for the instructions are given in ranges to reflect the effect of this early-out algorithm. Lower clock counts apply to smaller multipliers; larger clock counts apply to larger multipliers. The exact number of clocks can be calculated as follows: for multipliers that are 0, 1, 2, or 3, the instruction requires 7 clocks; for each two additional significant bits in the multiplier, add one clock.

See the MUL instruction.

Flags Changed:	AF	undefined
	CF	0 if conditions stated above are met, otherwise 1
	OF	0 if conditions stated above are met, otherwise 1
	PF	undefined
	SF	undefined
	ZF	undefined

IN—Input From I/O port

Instruction	Opcode	Action	Clocks (by mode):	rm, vm	pm
IN AL, imm8	E4	Load byte from port imm8 into AL		11	44
IN (E)AX, imm8	E5	Load word/dword from port imm8 into (E)AX		11	44
IN AL, DX	EC	Load byte from port specified by DX into AL		11	44
IN (E)AX, DX	ED	Load word/dword from port specified by DX into (E)AX		11	44

IN copies data from an I/O port, specified by the second operand, and stores it in a register, specified by the first operand. The port address can be specified either with an 8-bit immediate operand, which can address up to 256 ports, or with the 16-bit DX register, which can address the full 64kB range of ports.

Table A-8 shows which privilege-level checks are performed against the I/O protection level (IOPL) and the I/O permission bitmap (IOPB) for each mode.

Table A-8. *IN Privilege Level Checks*

	Protected Mode	Real Mode	Virtual-8086 Mode
IOPL	yes	yes	no
IOPB	yes (for 32-bit tasks)	no	yes

For I/O to succeed, the following must be true: In protected mode for 32-bit tasks, CPL must be ≤ IOPL, or the IOPB bit for the port must be cleared to 0; for 16-bit (80286) tasks, CPL must be ≤ IOPL, and the IOPB is not checked. In real mode, CPL must be ≤ IOPL, but since CPL is always 0, I/O always succeeds. In virtual-8086 mode, IOPL is never checked; only the IOPB bit for the port is checked. The IOPB bit must be cleared to 0.

For details, see the sections entitled "Protection Mechanisms" and "Other Processing Modes" in Chapter 4.

See the INS, INSB, INSD, and INSW instructions for string inputs.

Flags Changed: None

INC—Increment by One

Instruction	Opcode	Action	Clocks
INC r/m	FE /0 {8}, FF /0 {16, 32}	Increment r/m by 1	1/3
INC reg	40+reg {16, 32}	Increment reg by 1	1

INC increments its operand by 1. Unlike increments performed by the ADD instruction, INC does not modify the CF flag.

The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:

AF	0 if low nibble is nonzero, 1 if low nibble is zero
OF	0 if no overflow, 1 if overflow
PF	0 if odd parity, 1 if even parity
SF	high-order bit of result
ZF	0 if result was nonzero, 1 if result was zero

INS, INSB, INSD, and INSW—Input From I/O Port to String Element

Instruction	Opcode	Action	Clocks (by mode): rm, vm	pm
INSB	6C	Load byte from port DX and store at address in ES:[(E)DI]	12	44
INSD	6D	Load word from port DX and store at address in ES:[(E)DI]	12	44
INSW	6D	Load dword from port DX and store at address in ES:[(E)DI]	12	44

INS copies data from an I/O port, specified in the DX register, and stores it in a memory string, specified indirectly as the memory address contained in the ES:(E)DI register. Segment override prefixes are ignored for the destination address, which must always be relative to the ES segment.

If the DF flag is cleared to 0, the destination register is incremented by 1, 2, or 4 (depending on the operand size) to point to the next string element. If DF is set to 1, the destination register is decremented. The LOOP instruction or the REP instruction prefix can be used to repeat the operation.

In protected mode, the CPL must be less than or equal to the IOPL, and the IOPB bit for the port must be cleared to 0. In real mode, these I/O protections do not apply.

For details on privilege-level checking, see the description of the IN instruction.

The REP instruction prefix can be used to repeat the operation.

Flags Changed: None

INT *n*—Software Interrupts 0 to 2, or 5 to 255

Instruction	Opcode	Action	Clocks
INT imm8	CD	Generate interrupt number imm8	See below

INT generates a call to an interrupt or exception handler. The instruction operand is the interrupt vector, which is an offset into the IDT. In protected mode and virtual-8086 mode, the IDT contains segment descriptors for interrupt gates, trap gates, and/or task gates. In real mode, the IDT contains four-byte pointers. The clock counts are summarized below for all modes of operation.

In real mode, and in virtual-8086 mode when CPL ≤ IOPL, the number of clocks is 41*.

In non-task-switched protected mode, and in non-task-switched virtual-8086 mode when CPL > IOPL, the number of clocks is:

To same privilege level	131
To inner (more privilege) level	211
From virtual-8086 mode	220

In task-switched protected mode, and in task-switched virtual-8086 mode when CPL > IOPL, the number of clocks is as shown in Table A-9.

Table A-9. *INT n Clock Counts in Task-Switched Protected Mode and in Task-Switched Virtual-8086 Mode When CPL > IOPL*

	To Super386 Task	To 80286 Task	To Virtual-8086 Task
From Super386 task gate	427	366	468
From 80286 task gate	420	359	461

The DPL of the interrupt, trap, or task gate must be greater than (less privileged than) or equal to the CPL. The IF flag has no affect on software interrupts, although the flag is cleared to 0 after an interrupt through an interrupt gate (vs. a trap or task gate). If the interrupt handler is a procedure rather than a task, the processor pushes essential data (including the EFLAGS, CS, and EIP registers) on the stack before branching to the interrupt handler. The IRET or IRETD instruction is used to return from an interrupt or exception handler. If the interrupt is an NMI, additional NMIs are disabled until the handler returns.

See the section entitled "Interrupts and Exceptions" in Chapter 4 for details on the interrupt mechanism, interrupt handlers, and listings of the interrupt and exception vectors as defined in the Super386 architecture and for the IBM PC/AT.

For details on privilege-level checking, see the sections entitled "Protection Mechanisms" and "Other Processing Modes" in Chapter 4.

INT 3 and INT 4 are encoded as single-byte opcodes and are described separately.

Flags Changed:	IF	0 if an interrupt gate is accessed, otherwise unchanged
	TF	0
	NT	1 if nested task switch, otherwise unchanged

INT 3—Software Interrupt 3 (Breakpoint)

Instruction	Opcode	Action	Clocks
INT 3	CC	Generate interrupt number 3	See below

INT 3 is a one-byte instruction that generates an interrupt to vector 3 (breakpoint), providing an alternative method to using the debug registers. An unlimited number of breakpoints can be created with this instruction. The debug registers, by comparison, allow only a limited number of breakpoints, although they are more powerful. The debug registers must be used, however, for certain debugging such as in ROM-based programs, since the INT 3 opcode cannot replace an instruction in ROM. The clock counts are summerized below for all modes of operation.

In real mode and in virtual-8086 mode, the number of clocks is 41*.

In non-task-switched protected mode, the number of clocks is:

To same privilege level	131
To inner (more privileged) level	211
From virtual-8086 mode	220

In task-switched proteced mode, the number of clocks is as shown in Table A-10.

Table A-10. *INT 3 Clock Counts in Task-Switched Protected Mode*

	To Super386 Task	To 80286 Task	To Virtual-8086 Task
From Super386 task gate	427	366	468
From 80286 task gate	420	359	461

In all other respects, INT3 works identically in INT *n*.

Flags Changed:

IF	0 if an interrupt gate is accessed, otherwise unchanged
TF	0
NT	1 if nested task switch, otherwise unchanged

INTO—Software Interrupt 4 (Overflow)

Instruction	Opcode	Action	Clocks
INTO	CE	Generate interrupt number 4	See below

INTO 4 is a one-byte instruction that generates an interrupt to vector 4 if the OF flag is set to 1. If the OF flag is 0, the INTO instruction executes as a NOP. In all other respects, INT 4 works identically to INT *n*. The clock counts are shown below for all modes of operation.

In real mode and in virtual-8086 mode, the number of clocks is 43* if the OF flag is set to 1, or 4* if the OF flag is cleared to 0.

In non-task-switched protected mode, the number of clocks is:

To same privilege level	131
To inner (more privileged) level	211
From virtual-8086 mode	220

In task-switched proteced mode, the number of clocks is as shown in Table A-11.

Table A-11. *INTO Clock Counts in Task-Switched Protected Mode*

	To Super386 Task	To 80286 Task	To Virtual-8086 Task
From Super386 task gate	427	366	468
From 80286 task gate	420	359	461

Flags Changed:

IF	0 if an interrupt gate is accessed, otherwise unchanged
TF	0
NT	1 if nested task switch, otherwise unchanged

IRET and IRETD—Return from Interrupt Procedure or Task

Instruction	Opcode	Action	Clocks
IRET	CF	Return from interrupt, word pop	See Table A-12
IRETD	CF	Return from interrupt, dword pop	See Table A-12

The IRET and IRETD instructions are used to return from an interrupt procedure, or (in protected mode) from a interrupt-handling task. The value of the nested task flag in protected mode determines whether the return is to a procedure (NT = 0) or a task (NT = 1). The location to which control returns is determined by the type of interrupt or exception that occurred (interrupt, fault, trap, or abort). See the section entitled "Interrupts and Exceptions" in Chapter 4 for details. The clock counts are summerized below for all modes of operation.

In real mode, and in virtual-8086 mode when CPL ≤ IOPL, the number of clocks is 20*.

In non-task-switched protected mode, and in non-task-switched virtual-8086 mode when CPL > IOPL, the number of clocks is:

To same privilege level	91
To inner (more privileged) level	169
To virtual-8086 mode	110

In task-switched protected mode, and in task-switched virtual-8086 mode when CPl > IOPL, the number of clocks is as shown in Table A-12.

Table A-12. *IRET and IRETD Clock Counts in Task-Switched Protected Mode and in Task-Switched Virtual-8086 Mode When CPL > IOPL*

	To Super386 Task	To 80286 Task	To Virtual-8086 Task
From Super386 task gate	446	385	487
From 80286 task gate	439	378	480

When returning, IRET pops word operands from the stack, whereas IRETD pops dword operands. IRET and IRETD are similar to the far return in real and virtual-8086 modes, except that IRET and IRETD pop the EFLAGS register in addition to the CS and EIP registers. If the return is to a procedure with less privilege (higher DPL), IRET and IRETD also pop the stack segment selector and stack pointer for the less privileged procedure.

see p. 4-88 for image

Privilege-level checks are made during the return. For returns from interrupt-handling procedures, the RPL of the destination code segment selector must be greater than or equal to the CPL. For returns from interrupt-handling tasks, the DPL of the destination TSS or of the task gate (if used) must be greater than or equal to both the CPL and the RPL. If the interrupt handler services an NMI, additional NMIs are disabled until the handler is exited through an IRET or IRETD.

For details on privilege-level checking, see the sections entitled "Protection Mechanisms" and "Other Processing Modes" in Chapter 4.

Flags Changed: When handler is a procedure—Restored from EFLAGS of prior procedure's stack, except that IOPL is restored only if CPL = 0, and the RF and VM flags are restored only with the IRETD instruction.

When handler is a task—Restored from EFLAGS of prior task's TSS.

Jcc—Conditional Jump

Instruction	Opcode	Action	Clocks
JE/JZ rel	0F 84	Jump near by displacement rel if ZF = 1	See Table A-13
JE/JZ rel8	74	Jump short by displacement rel8 if ZF = 1	See Table A-13
JNE/JNZ rel	0F 85	Jump near by displacement rel if ZF = 0	See Table A-13
JNE/JNZ rel8	75	Jump short by displacement rel8 if ZF = 0	See Table A-13
JA/JNBE rel	0F 87	Jump near by displacement rel if CF = 0 and ZF = 0	See Table A-13
JA/JNBE rel8	77	Jump short by displacement rel8 if CF = 0 and ZF = 0	See Table A-13
JBE/JNA rel	0F 86	Jump near by displacement rel if CF = 1 or ZF = 1	See Table A-13
JBE/JNA rel8	76	Jump short by displacement rel8 if CF = 1 or ZF = 1	See Table A-13
JB/JNAE rel	0F 82	Jump near by displacement rel if CF = 1	See Table A-13
JB/JNAE rel8	72	Jump short by displacement rel8 if CF = 1	See Table A-13
JAE/JNB rel	0F 83	Jump near by displacement rel if CF = 0	See Table A-13
JAE/JNB rel8	73	Jump short by displacement rel8 if CF = 0	See Table A-13
JG/JNLE rel	0F 8F	Jump near by displacement rel if ZF = 0 and SF = OF	See Table A-13
JG/JNLE rel8	7F	Jump short by displacement rel8 if ZF = 0 and SF = OF	See Table A-13
JGE/JNL rel	0F 8D	Jump near by displacement rel if SF = OF	See Table A-13
JGE/JNL rel8	7D	Jump short by displacement rel8 if SF = OF	See Table A-13
JL/JNGE rel	0F 8C	Jump near by displacement rel if SF <> OF	See Table A-13
JL/JNGE:rel8	7C	Jump short by displacement rel8 if SF <> OF	See Table A-13
JLE/JNG rel	OF 8E	Jump near by displacement rel if ZF = 1 or SF <> OF	See Table A-13
JLE/JNG rel8	7E	Jump short by displacement rel8 if ZF = 1 or SF <> OF	See Table A-13
JS rel	0F 88	Jump near by displacement rel if SF = 1	See Table A-13
JS rel8	78	Jump short by displacement rel8 if SF = 1	See Table A-13
JNS rel	OF 89	Jump near by displacement rel if SF = 0	See Table A-13
JNS rel8	79	Jump short by displacement rel8 if SF = 0	See Table A-13
JO rel	0F 80	Jump near by displacement rel if OF = 1	See Table A-13
JO rel8	70	Jump short by displacement rel8 if OF = 1	See Table A-13
JNO rel	0F 81	Jump near by displacement rel if OF = 0	See Table A-13
JNO rel8	71	Jump short by displacement rel8 if OF = 0	See Table A-13
JP rel	OF 8A	Jump near by displacement rel if PF = 1	See Table A-13
JP rel8	7A	Jump short by displacement rel8 if PF = 1	See Table A-13
JNP rel	OF 8B	Jump near by displacement rel if PF = 0	See Table A-13
JNP rel8	7B	Jump short by displacement rel8 if PF = 0	See Table A-13
JCXZ rel8	E3	Jump short by displacement rel8 if register CX = 0	See Table A-13
ECXZ rel8	E3	Jump short by displacement rel8 if register ECX = 0	See Table A-13

The Jcc instructions cause a near jump (16-bit or 32-bit operand) or short jump (8-bit operand) to the location, within the current code segment, that is specified by the operand. The operand is an offset from the address in the EIP register. If the operand size is 16, only the low word of the EIP register is used to obtain the address displacement.

In protected, real, and virtual-8086 modes, the number of clock cycles required for the execution of jump instructions depends on the type of processor, size of displacement, whether or not the jump is taken, and whether there is a cache hit, as shown in Table A-13.

Table A-13. *Jcc Clock Counts*

Jump		8-bit Displacement	16-bit or 32-bit Displacement
38605 Processor	Jump taken, cache hit	2	6*[1]
	Jump taken, cache miss	5	6*
	Jump not taken	1	1*
38600 Processor	Jump taken	5	6*
	Jump not taken	1	1*

1 See "Clock Counts" in this appendix for an explanation of *.

Unlike CALL instructions, jump instructions do not push anything onto the stack in anticipation of a return. There is no operation (other than another jump) that causes a return.

The instruction tests the flags specified on page A-59 and transfers control if the flag conditions are met. If flag conditions are not met, the jump instruction is ignored and the program continues execution at the next instruction. The fastest jump execution occurs in short jumps (within ±128 bytes of the next instruction) in the current code segment.

Some opcodes have more than one mnemonic because their effects can be interpreted different ways. In the mnemonics listing, the following abbreviations are used:

A	above (for comparing unsigned integers)
B	below (for comparing unsigned integers)
C	carry
CX	CX register
E	equal to
ECX	ECX register
G	greater than (for comparing signed integers)
L	less than (for comparing signed integers)
N	not
O	overflow
P	parity
PE	parity even
PO	parity odd
S	sign
Z	zero.

To branch conditionally to a location in a different code segment, use the complementary sense of the Jcc instruction, then use an unconditional far jump to the other segment. The JCXZ and JECXZ instructions are used at the start of conditional loops that end in conditional loops, to avoid executing the loops unnecessarily if there is a zero in the CX or ECX register.

Jumps flush the instruction pipeline.

Flags Changed: None, if there is no task switch. See "JMP (task)."

JMP (near)—Jump Within Same Segment

Instruction	Opcode	Action	Clocks
JMP rel	E9	Jump near by offset rel	See Table A-14
JMP rel8	EB	Jump short by offset rel8	See Table A-14
JMP [r/m]	FF /4	Jump near by offset in [r/m]	8/10

A near JMP transfers control to the location, within the current code segment, specified by the operand. The operand specifies an offset from the EIP either directly (the offset is the operand itself) or indirectly (the offset is contained in a register or memory location). In the direct form of the instruction, the operand size it determined by the code segment. If the operand size is 16, the processor clears to 0 the upper word of the new EIP to enable a 32-bit jump to follow.

The number of clock cycles required in all operating modes for execution of the first two forms of the instruction depends on the whether or not the jump is taken and whether there is a cache hit, as shown in Table A-14.

Table A-14. *Near JMP rel and rel8 Clock Counts*

Jump	JMP rel	JMP rel8
Jump not taken	1	1
Jump taken, cache hit	6	2
Jump taken, cache miss	6	5

Unlike CALL instructions, jump instructions do not push anything onto the stack in anticipation of a return. There is no operation (other than another jump) that causes a return.

Use the far JMP instruction when branching to a code segment that differs from the current code segment. See the separate description of the task JMP, which is the same opcode as the far JMP.

Jumps (near or far) flush the instruction pipeline.

Flags Changed: None

JMP (far)—Jump to Different Segment

Instruction	Opcode	Action	Clocks (by mode)	rm, vm	pm
JMP sel:off	EA	Jump far to address sel:off		13	See below
JMP [m]	FF /5	Jump far to address in [m]		17	See below

The far JMP branches to a location in a different code segment than the current code segment. (The instruction can also be used for branches within the current code segment, but the near jump executes faster for such jumps.) The operand specifies a far pointer of 48 bits or 32 bits, depending on operand size. The pointing is either direct (the pointer is the operand itself) or indirect (the pointer is contained in a register or memory location).

In protected mode, the number of clocks required depends on the destination of the jump, as follows:

To a code segment	46
To a call gate	61

In the direct form of the instruction, the operand size is determined by the code segment. For both direct and indirect forms, if the operand size is 16, the processor clears to 0 the upper word of the new EIP. Unlike CALL instructions, jump instructions do not push anything onto the stack in anticipation of a return.

In protected mode, the instruction uses the segment selector as an offset into a descriptor table. The descriptor to which the segment selector points may directly specify a code segment, or it may specify a call gate, a task gate, or a task state segment. When the selector references a call gate, the call gate selector and offset are used for the address, and the offset value in the instruction is ignored. Call gates are described in the section entitled "Control Gates and System Calls" in Chapter 4.

For details on privilege-level checking, see the sections entitled "Protection Mechanisms" and "Other Processing Modes" in Chapter 4.

In real or virtual-8086 mode, the far pointer contains the new CS selector and (E)IP value.

Jumps (near or far) flush the instruction pipeline.

Flags Changed: None

JMP (task)—Jump to Different Task (Switch Task)

Instruction	Opcode	Action	Clocks
JMP sel:off	EA	Jump to task at address sel:off	See Table A-15
JMP [m]	FF /5	Jump to task at address in [m]	See Table A-15

A task JMP instruction works like a protected-mode far JMP, except that the task JMP selector specifies a TSS descriptor or a task gate descriptor, which in turn specifies a TSS descriptor. See the description of far JMP.

The number of clock cycles required for execution of the instruction depends on the source and destination of the jump, as shown in Table A-15.

Table A-15. *Task JMP Clock Counts*

	To Super386 Task	To 80286 Task	To Virtual-8086 Task
From Super386 task	438	377	479
From 286 task	431	370	472

If the new TSS descriptor is not busy, the current task state is saved in the existing TSS, and the new task state is loaded. The segment selector of the old TSS is saved in the back-link field of the new TSS, and the nested task (NT) flag is set to 1 in the new TSS. The section entitled "Multitasking" in Chapter 4 describes this mechanism, task gates, and TSSs.

In real mode, the selector provided in the operand does not refer to a segment descriptor. Instead, the selector is simply shifted left four bits and written into the descriptor-base field of the segment's shadow register.

Flags Changed: All

LAHF—Load Flags Into AH Register

Instruction	Opcode	Action	Clocks
LAFH	9F	Load low byte of flags word into AH	2*

LAHF copies the low-order byte of the EFLAGS register into the AH register. The resulting bits in the AH register are:

AH bit 7	SF
AH bit 6	ZF
AH bit 5	0
AH bit 4	AF
AH bit 3	0
AH bit 2	PF
AH bit 1	1
AH bit 0	CF

Flags Changed: None

LAR—Load Access Rights Byte

Instruction	Opcode	Action	Clocks
LAR r, r/m	0F 02	Load access rights part of r/m to r	25*/28*

LAR loads the first operand with the access rights of the segment referenced by the second operand (a segment selector). The instruction allows examination of a segment descriptor's access rights without revealing the physical address of the descriptor's base.

LAR copies the high dword of the two-dword segment descriptor referenced by the selector and masks (ANDs) it with the value 00FxFF00. If the descriptor can be read, and it is of the proper type, the result is stored in the first operand and the ZF flag is set to 1.

Masking prevents the base portion of the upper dword (bits 31:24 and 7:0) from being seen and leaves the limit portion (bits 19:16) undefined. More important, masking makes visible the access-rights bits defined in Table A-16:

Table A-16. *LAR Access-Rights Bit Definitions*

Bit Description	Bit Number
Type field	12:8
Descriptor privilege level (DPL)	14:13
Present bit	15
Available bit	20 (for dword operands)
Default size/upper bound bit	22 (for dword operands)
Granularity bit	23 (for dword operands)

LAR operates on all code segments, data segments, TSSs, call gates, and task gates—both 16-bit and 32-bit—but not on trap gates or interrupt gates. In protected mode, LAR executes at all privilege levels. In real or virtual-8086 modes, LAR generates an invalid-opcode fault (exception 6).

Flags Changed: ZF = 1 if the selector is visible and of the right type, otherwise 0.

LDS—Load Pointer into DS and a Register

Instruction	Opcode	Action	Clocks (by mode):	rm, vm	pm
LDS r, m	C5 {16, 32}	Load pointer from m into DS:r		11*	25*

LDS loads a far pointer (segment selector and offset) into the DS segment selector register and a general purpose register. The pointer is copied from the memory location specified by the second (source) operand. The 16-bit segment selector portion of the pointer is loaded into the DS register. The 16-bit or 32-bit offset is loaded into the register specified by the first (destination) operand.

The size of the destination register is determined by the operand-size attribute. The processor loads the segment descriptor into the segment selector's shadow register when the segment selector is loaded.

Also see the LES, LSS, LFS, and LGS instructions.

Flags Changed: None

LEA—Load Effective Address

Instruction	Opcode	Action	Clocks
LEA r, m	8D	Load effective address for m in r	1 (2 if index is included)

LEA calculates the effective address (segment offset) of the second operand and stores it in the first operand.

The instruction simply calculates the address; it does not make an actual memory reference or check the validity of the address. The instruction uses the same MODr/m-byte (and optionally, SIB-byte) encoding of other instructions that generate effective addresses from a base, index, displacement, and scaling factor. Segment override prefixes in the instruction are ignored. If the operand size is less than the address size, only the low-order bits of the offset are stored. If the address size is less than the operand size, the offset value is zero-extended.

While the instruction is normally used to determine effective addresses, it can also be used for a variety of register arithmetic. For example, it can be used to fill the destination register with an immediate operand or with the sum of a base register plus an index register. For this, the MODr/m byte of the instruction is selected to specify only the parts of the effective address that are needed for the arithmetic, and the second operand must have the form of a memory operand. The destination register will be filled with the sign-extended result. In these applications, the instruction differs from the ADD instruction in that the flags are not altered.

Flags Changed: None

LEAVE—Leave Nested Procedure

Instruction	Opcode	Action	Clocks
LEAVE	C9	Set (E)SP value to (E)BP; pop frame pointer into (E)BP	4*

LEAVE reverses the action of its corresponding ENTER instruction. LEAVE assigns the value of the (E)BP register to the (E)SP register, thereby releasing the stack frame generated by ENTER. The top of the stack then contains the caller's (E)BP, which is popped into (E)BP.

By contrast, a RET instruction pops all of the parameters pushed on the stack by the procedure that is being left.

LEAVE assumes the operand size of the code segment in which it resides.

Flags Changed: None

LES—Load Pointer Into ES and a Register

Instruction	Opcode	Action	Clocks (by mode):	rm, vm	pm
LES r, m	C4	Load pointer from m into ES:r		11*	25*

LES loads a far pointer (segment selector and offset) into the ES segment selector register and a general purpose register. The pointer is copied from the memory location specified by the second (source) operand. The 16-bit segment selector portion of the pointer is loaded into the ES register. The 16-bit or 32-bit offset is loaded into the register specified by the first (destination) operand.

The size of the destination register is determined by the operand-size attribute. The processor loads the segment descriptor into the segment selector's shadow register when the segment selector is loaded.

Also see the LDS, LSS, LFS, and LGS instructions.

Flags Changed: None

LFS—Load Pointer Into FS and a Register

Instruction	Opcode	Action	Clocks (by mode):	rm, vm	pm
LFS r, m	0F B4	Load pointer from m into FS:r		11*	25*

LFS loads a far pointer (segment selector and offset) into the FS segment selector register and a general purpose register. The pointer is copied from the memory location specified by the second (source) operand. The 16-bit segment selector portion of the pointer is loaded into the FS register. The 16-bit or 32-bit offset is loaded into the register specified by the first (destination) operand.

The size of the destination register is determined by the operand-size attribute. The processor loads the segment descriptor into the segment selector's shadow register when the segment selector is loaded.

See the LDS, LES, LSS, and LGS instructions.

Flags Changed: None

LGDT—Load Global Descriptor Table

Instruction	Opcode	Action	Clocks
LGDT m	0F 01 /2	Load global descriptor table register from m	12*

LGDT initializes the GDT by loading the GDT register (GDTR) from a six-byte memory location specified by the instruction's operand.

For 32-bit operands, a two-dword memory structure is used. The first dword begins with a word for the segment limit, which is followed by the low-order word of the segment base. The second dword contains the high-order word of the segment base; the upper word of the second dword is undefined.

For 16-bit operands, a three-word memory structure is used. The first word is the segment limit. The second word is the low-order word of the segment base. The first byte of the third word is the high-order byte of the segment base; the upper byte of the third word is undefined.

See the section entitled "Descriptor Tables and Their Registers" in Chapter 4 for details.

Unlike the SGDT instruction, LGDT is a privileged instruction and can only be used from privilege level 0.

Flags Changed: None

LGS—Load Pointer Into GS and a Register

Instruction	Opcode	Action	Clocks (by mode):	rm, vm	pm
LGS r, m	0F B5	Load pointer from m into GS:r		11*	25*

LGS loads a far pointer (segment selector and offset) into the GS segment selector register and a general purpose register. The pointer is copied from the memory location specified by the second (source) operand. The 16-bit segment selector portion of the pointer is loaded into the GS register. The 16-bit or 32-bit offset is loaded into the register specified by the first (destination) operand.

The size of the destination register is determined by the operand-size attribute. The processor loads the segment descriptor into the segment selector's shadow register when the segment selector is loaded.

Also see the LDS, LSS, LES, and LFS instructions.

Flags Changed: None

LIDT—Load Interrupt Descriptor Table

Instruction	Opcode	Action	Clocks
LIDT m	0F 01 /3	Load interrupt descriptor table register from m	12*

LIDT initializes the IDT by loading the IDT register (IDTR) from a six-byte memory location specified by the instruction's operand.

For 32-bit operands, a two-dword memory structure is used. The first dword begins with a word for the segment limit, which is followed by the low-order word of the segment base. The second dword contains the high-order word of the segment base. The upper word of the second dword is undefined.

For 16-bit operands, a three-word memory structure is used. The first word is the segment limit. The second word is the low-order word of the segment base. The first byte of the third word is the high-order byte of the segment base; the upper byte of the third word is undefined.

See the section entitled "Descriptor Tables and Their Registers" in Chapter 4 for details.

Unlike the SIDT instruction, LIDT is a privileged instruction and can only be used from privilege level 0.

Flags Changed: None

LLDT—Load Local Descriptor Table

Instruction	Opcode	Action	Clocks
LLDT r/m16	0F 00 /2	Load local descriptor table register from r/m16	27*/28*

LLDT loads the local descriptor table register (LDTR) with the segment selector located in the instruction's operand. The selector must point to an LDT segment descriptor in the GDT. The field reserved for the LDT selector in the TSS is not affected by this instruction.

LDTs are only used in protected mode. If the LDTR is loaded with a null (zero) selector, references to the LDT descriptor generate a general-protection fault, except references by the LSL, LAR, VERR, or VERW instructions.

The instruction can only be executed at privilege level 0. See the SLDT instruction.

Flags Changed: None

LMSW—Load Machine Status Word

Instruction	Opcode	Action	Clocks
LMSW r/m16	0F 01 /6	Load machine status word from r/m16	19*/20*

LMSW is provided for compatibility with the 80286. It is not recommended for use in new Super386 code. Instead, the MOV CR0, rr instruction should be used.

LMSW copies the operand into the lower word of control register CR0, the machine status word (MSW). The instruction must be followed by a jump or call to flush the instruction pipeline. While the processor can be switched from real mode to protected mode with LMSW, it cannot be switched back to real mode with LMSW; this must be done with MOV CR0.

The instruction can only be executed at privilege level 0. See the SMSW instruction.

Flags Changed: None

LOCK—Lock Memory Bus (Instruction Prefix)

Instruction	Opcode	Action	Clocks
LOCK	F0	Activate LOCK signal for subsequent instruction	0*

The LOCK instruction prefix provides secure access to memory locations. It prevents access by other operations to the memory operand of the associated instruction. The lock remains enabled for the duration of the instruction.

The LOCK prefix can be decoded independently of normal instruction pipeline operations. A delay of one clock occurs if the locked instruction either follows a one-clock instruction or is the target of a jump.

Memory accesses with the following instructions may be prefixed by LOCK: ADD, ADC, AND, BTC, BTR, BTS, DEC, INC, NEG, NOT, OR, SBB, SUB, XCHG, and XOR.

The XCHG instruction asserts a bus lock signal whether the instruction is preceded by a LOCK prefix or not. Misalignment of memory operands does not affect the lock operation.

Flags Changed: None

LODSB, LODSW, and LODSD—Load String Operands

Instruction	Opcode	Action	Clocks
LODSB	AC	Copy byte at address in DS:[(E)SI] to AL	6
LODSW	AD	Copy word at address in DS:[(E)SI] to AX	6
LODSD	AD	Copy dword at address in DS:[(E)SI] to EAX	6

The LODS instructions copy the operand at the memory location (source operand) found in the DS:(E)SI register into a register. LODSB, LODSW, and LODSD copy the source into the AL, AX, and EAX registers, respectively.

If the DF flag is cleared to 0, the source register is incremented by 1, 2, or 4 (depending on operand size) to point to the next string element. If DF is set to 1, the source register is decremented. The LOOP instruction or the REP instruction prefix can be used to repeat the operation.

Offset (E)SI is referenced to the DS segment register, unless a segment override prefix changes this default.

Flags Changed: None

LOOP and LOOPcc—Loop Control with CX Counter

Instruction	Opcode	Action	Clocks
LOOP rel8	E2	Decrement (E)CX; jump short by displacement rel8 if (E)CX <> 0	3/8
LOOPNE/LOOPNZ rel8	E0	Decrement (E)CX; jump short by displacement rel8 if (E)CX <> 0 and ZF = 0	3/8
LOOPE/LOOPZ rel8	E1	Decrement (E)CX; jump short by displacement rel8 if (E)CX <> 0 and ZF = 1	3/8

The LOOP instructions decrement the (E)CX register and check certain conditions. If the conditions are all met, control transfers to the displacement specified by the operand.

The number of clock cycles required for execution of loop instructions depends on whether or not the loop is taken, as follows:

Loop taken	8
Loop not taken	3

The conditions for the loop are listed in Table A-17. The value of (E)CX is the value after being decremented by 1 at the beginning of the operation.

Table A-17. *Loop Conditions*

Instruction	Value of (E)CX	ZF flag
LOOP	<> 0	0 or 1
LOOPNE/LOOPNZ	<> 0	0
LOOPE/LOOPZ	<> 0	1

To code an iteration, put the LOOP instruction at the bottom of the loop and a label for the operand (loop destination) at the top of the loop. Load the counter with an unsigned integer. All LOOP instructions assume the operand-size and address-size attributes from their related code segment, although they can be overridden with instruction prefixes.

Flags Changed: None

LSL—Load Segment Limit

Instruction	Opcode	Action	Clocks
LSL r, r/m	0F 03	Load r with segment limit for selector in r/m	23*/26*

LSL loads the first operand with the limit for the segment whose selector is given in the second operand.

The segment limit is copied and rearranged from the descriptor for the segment. The resulting 32-bit limit is contiguously assembled and, if necessary, shifted so that it is byte-granular. If the destination register is 16-bits, the limit is truncated to its low-order 16 bits. The LAR instruction can be used to determine whether a segment is expand-up or expand-down.

If the selector is visible and accessible according to the protection rules, the ZF flag is set to 1. If the selector is not visible or accessible, or if the selector does not contain a limit field, the ZF flag is cleared to 0 and the destination register is not modified. See the sections entitled "Segmentation" and "Protection Mechanisms" in Chapter 4.

LSL operates on all code segments, data segments, and TSSs—both 16-bit and 32-bit—but not on call gates, trap gates, interrupt gates, or task gates. It can only be used in protected mode.

Flags Changed: ZF = 1 if the selector is visible and of the right type, otherwise 0.

LSS—Load Pointer Into SS and a Register

Instruction	Opcode	Action	Clocks (by mode):	rm, vm	pm
LSS r, m	0F B2	Load pointer from m into SS:r		11*	24*

LSS copies a far pointer (segment selector and offset) stored at the memory address given in the second operand, loads the selector part into the SS segment register, and loads the offset part into the register specified by the first operand.

The size of the first operand (destination register) is determined by the operand-size attribute. Dword operands have six-byte pointers (16-bit selector and 32-bit offset), and word operands have four-byte pointers (16-bit selector and 16-bit offset).

LSS is used to load SS and ESP simultaneously during the initialization of a new stack, replacing a sequence of two MOVs. Also see the LDS, LSS, LES, and LFS instructions.

Flags Changed: None

LTR—Load Task Register

Instruction	Opcode	Action	Clocks
LTR r/m16	0F 00 /3	Copy r/m16 operand to task register	33*/34*

What initiates the task switch?

LTR loads its operand, a selector for a TSS, into the Task Register (TR). The TSS is then marked busy, but a task switch is not initiated. The selector must reference a TSS descriptor in the GDT.

This instruction operates only at privilege level 0. A general-protection fault (exception 13) occurs if the TSS is already busy.

Flags Changed: None

MOV—Copy Data From/To General Registers

Instruction	Opcode	Action	Clocks
MOV r, r/m	8A {8}, 8B {16, 32}	Copy value in r/m to r	1/2
MOV r/m, r	88 {8}, 89 {16, 32}	Copy value in r to r/m	1/2
MOV r/m, imm	C6 {8}, C7 {16, 32}	Copy imm value to r/m	2/4
MOV reg, imm	B0+reg {8}, B8+reg {16, 32}	Copy imm value to reg	1
MOV AL, moff	A0 {8}	Copy value at moff to AL	2
MOV (E)AX, moff	A1 {16, 32}	Copy value at moff to (E)AX	2
MOV moff, AL	A2 {8}	Copy value in AL to moff	2
MOV moff, (E)AX	A3 {16, 32}	Copy value in (E)AX to moff	2

This form of MOV copies the second operand to the first operand. The operands must be the same size. To copy values of different sizes, use MOVSX or MOVZX. Also see the other forms of MOV.

The acronym MOV is a misnomer for this instruction; the operation is a copy, not a move.

Flags Changed: None

MOV—Load Segment Registers

Instruction	Opcode	Action	Clocks (by mode):	rm, vm	pm
MOV DS, r/m	8E /3	Copy value in r/m to DS		6/8	23/25
MOV SS, r/m	8E /2	Copy value in r/m to SS		6/8	23/25
MOV ES, r/m	8E /0	Copy value in r/m to ES		6/8	23/25
MOV FS, r/m	8E /4	Copy value in r/m to FS		6/8	23/25
MOV GS, r/m	8E /5	Copy value in r/m to GS		6/8	23/25

This form of MOV initializes a new segment so that it can be addressed. The instruction copies the second operand, a 16-bit segment selector, to the first operand, a segment selector register. The operation also causes the processor to load the segment descriptor referenced by the selector into the selector's shadow register. If the second operand is a dword, its upper word is disregarded.

After stack segment loads, hardware interrupts (including NMI) are inhibited during the next instruction. This allows the next instruction to load the ESP register. The sequence should therefore be:

```
MOV SS r/m
MOV ESP top_of_stack
```

For data segment loads (except stack segments), a general-protection fault (exception 13) is generated if the descriptor's DPL is less than the maximum of CPL and the selector's RPL. For stack segment loads, a stack fault (exception 12) is generated if the descriptor's DPL, the selector's RPL, and the CPL are not all equal. The DS and ES registers can be loaded with a null selector. An access to a segment with a null selector will generate a general-protection fault (exception 13).

An invalid-opcode fault (exception 6) is generated if an attempt is made to load the CS segment register, since this would result in a CS value that is unrelated to its associated EIP value. For code segment changes, the CS and EIP registers must be loaded simultaneously. This is done with a far jump or call, return from call, interrupt or exception, or task switch.

For details on privilege-level checking, see the sections entitled "Protection Mechanisms" and "Other Processing Modes" in Chapter 4. Also see the several other forms of MOV.

MOV is a misnomer for this instruction; the operation is a copy, not a move.

Flags Changed: None

MOV—Store Segment Register

Instruction	Opcode	Action	Clocks
MOV r/m, CS	8C /1	Copy value in CS to r/m	2/3
MOV r/m, DS	8C /3	Copy value in DS to r/m	2/3
MOV r/m, SS	8C /2	Copy value in SS to r/m	2/3
MOV r/m, ES	8C /0	Copy value in ES to r/m	2/3
MOV r/m, FS	8C /4	Copy value in FS to r/m	2/3
MOV r/m, GS	8C /5	Copy value in GS to r/m	2/3

This form of MOV copies the second operand, a 16-bit segment selector, to the first operand. If the first operand is a dword, its upper word is filled with zeros.

The instruction can be used for transfers between segment registers, in which case it causes the processor to load the segment descriptor for the selector into the segment shadow register.

See the several other forms of MOV.

Flags Changed: None

MOV—Load Control, Debug, or Test Registers

Instruction	Opcode	Action	Clocks
MOV CR0, r32	0F 22 /0	Copy value in r32 to CR0	16*
MOV CR2, r32	0F 22 /2	Copy value in r32 to CR2	5*
MOV CR3, r32	0F 22 /3	Copy value in r32 to CR3	110*
MOV DR0, r32	0F 23 /0	Copy value in r32 to DR0	19*
MOV DR1, r32	0F 23 /1	Copy value in r32 to DR1	19*
MOV DR2, r32	0F 23 /2	Copy value in r32 to DR2	19*
MOV DR3, r32	0F 23 /3	Copy value in r32 to DR3	19*
MOV DR6, r32	0F 23 /6	Copy value in r32 to DR6	10*
MOV DR7, r32	0F 23 /7	Copy value in r32 to DR7	16*
MOV TR6, r32	0F 26 /6	Copy value in r32 to TR6	13*
MOV TR7, r32	0F 26 /7	Copy value in r32 to TR7	5*

This form of MOV copies the second operand into the first operand, which is a control register, debug register, or test register. The dword size of the second operand is not affected by the operand-size attribute.

The control register CR0 stores the machine status word, and the processor mode can be changed with MOV CR0 r32, followed by a jump or call to clear the instruction pipeline. This instruction should be used rather than LMSW.

For details of control, debug, and test register usage, see the sections entitled "System Register," "Debugging," and "Testing the TLB" in Chapter 4.

This instruction operates only at privilege level 0. See the several other forms of MOV.

The acronym MOV is a misnomer; the operation is a copy, not a move.

Flags Changed:

AF	undefined
CF	undefined
OF	undefined
PF	undefined
SF	undefined
ZF	undefined

MOV—Store Control, Debug, or Test Registers

Instruction	Opcode	Action	Clocks
MOV r32, cr	0F 20	Copy value in cr to r32	2*
MOV r32, dr	0F 21	Copy value in dr to r32	2* to 6*
MOV r32, tr	0F 24	Copy value in tr to r32	2*

This form of MOV copies the second operand (a control, debug, or test register) to the first operand, a general register. The dword size of the second operand is not affected by the operand-size attribute.

The range of clocks shown for storing debug registers is due to uncertainty about external bus activity. If the bus is free, the instruction will execute in the least number of clocks. If the bus is being used when the instruction executes, the operation will take longer.

A debug fault or trap (exception 1) is generated if the GD bit (bit 13) in DR7 is set to 1 and an attempt is made to access one of the debug registers. See the section entitled "Debugging" in Chapter 4.

This instruction operates only at privilege level 0. Also see the several other forms of MOV.

Flags Changed:

AF	undefined
CF	undefined
OF	undefined
PF	undefined
SF	undefined
ZF	undefined

MOVS, MOVSB, MOVSW, and MOVSD—Copy String Data

Instruction	Opcode	Action	Clocks
MOVSB	A4	Copy byte at address in DS:[(E)SI] to byte at address in ES:[(E)DI]	8
MOVSW	A5	Copy word at address in DS:[(E)SI] to word at address in ES:[(E)DI]	8
MOVSD	A5	Copy dword at address in DS:[(E)SI] to dword at address in ES:[(E)DI]	8

This form of MOV copies data from the memory address contained in the second register, DS:(E)SI, to the memory address contained in the first register, ES:(E)DI. A segment override prefix can be used on the source segment (DS) but not on the destination segment (ES).

If the DF flag is cleared to 0, the source and destination registers are incremented by 1, 2, or 4 (depending on operand size) to point to the next string element. If DF is set to 1, the registers are decremented. The LOOP instruction or the REP instruction prefix can be used to repeat the operation.

Also see the several other forms of MOV.

Flags Changed: None

MOVSX—Copy Data With Sign Extension

Instruction	Opcode	Action	Clocks
MOVSX r, r/m8	0F BE	Copy and sign-extend value in r/m8 as word/dword to r	2*/3*
MOVSX r, r/m16	0F BF	Copy and sign-extend value in r/m16 as dword to r	2*/3*

MOVSX copies the value addressed by the second operand, sign-extends it, and stores it in the first operand. MOVSX sign-extends the second operand, which is a byte or word, to a word or dword.

Also see the several other forms of MOV.

Flags Changed: None

MOVZX—Copy Data With Zero Extend

Instruction	Opcode	Action	Clocks
MOVZX r, r/m8	0F B6	Copy and zero-extend value in r/m8 as word/dword to r	2*/3*
MOVZX r32, r/m16	0F B7	Copy and zero-extend value in r/m16 as dword to r32	2*/3*

MOVZX copies the value addressed by the second operand, zero-extends it, and stores it in the first operand. MOVZX zero-extends the second operand, which is a byte or word, to a word or dword.

Also see the several other forms of MOV.

Flags Changed: None

MUL—Unsigned Integer Multiplication

Instruction	Opcode	Action	Clocks
MUL AL, r/m8	F6 /4	Multiply AL by r/m8; result in AX	10 to 14
MUL AX, r/m16	F7 /4	Multiply AX by r/m16; result in DX:AX	10 to 18
MUL EAX, r/m32	F7 /4	Multiply EAX by r/m32; result in EDX:EAX	10 to 26

MUL multiplies its two operands and stores the result in the AX, DX:AX, or EDX:EAX register, depending on the size of the first operand. For word multiplication, the DX register contains the high-order word of the result. For dword multiplication, the EDX register contains the high-order word of the result.

Clock counts for the instructions are given in ranges to reflect the effect of this early-out algorithm. Lower clock counts apply to smaller multipliers, and larger clock counts apply to larger multipliers. The exact number of clocks can be calculated as follows: for multipliers that are 0, 1, 2, or 3, the instruction requires 7 clocks; for each two additional significant bits in the multiplier, add one clock.

All values are treated as unsigned integers. Also see the IMUL instruction for signed multiplication.

Flags Changed:

AF	undefined
CF	0 if upper half of result is 0; otherwise 1
OF	0 if upper half of result is 0; otherwise 1
PF	undefined
SF	undefined
ZF	undefined

NEG—Negate Using Two's Complement

Instruction	Opcode	Action	Clocks
NEG r/m	F6 /3 {8}, F7 /3 {16, 32}	Negate r/m (two's complement method)	1/5

NEG subtracts its operand from 0, using the two's complement method. The result replaces the original operand.

The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:	CF	0 if result was zero, 1 if result was nonzero
	OF	0 if no overflow, 1 if overflow
	PF	0 if odd parity, 1 if even parity
	SF	high-order bit of result
	ZF	0 if result was nonzero, 1 if result was zero

NOP—No Operation

Instruction	Opcode	Action	Clocks
NOP	90	No operation	2*

NOP performs no operation, other than to increment the EIP. It can be used for delays in timing loops or to align labels to dword boundaries.

Flags Changed: None

NOT—Bitwise Complement

Instruction	Opcode	Action	Clocks
NOT r/m	F6 /2 {8}, F7 /2 {16, 32}	Negate r/m (one's complement method)	1/5

NOT replaces the operand with its one's complement.

In NOT operations, a 1 bit is written when the operand contains a 0, and a 0 bit is written when the operand contains a 1.

The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed: None

OR—Inclusive OR

Instruction	Opcode	Action	Clocks
OR r, r/m	0A {8}, 0B {16, 32}	Logical OR of r/m and r operands, result in r	1/5
OR r/m, r	08 {8}, 09 {16, 32}	Logical OR of r/m and r operands, result in r/m	1/5
OR r/m, imm	80 /1 {8}, 81 /1 {16, 32}	Logical OR of r/m and imm operands, result in r/m	1/5
OR r/m, imm8	83 /1 {16, 32}	Logical OR of r/m and imm8 operands, result in r/m	1/5
OR AL, imm8	0C {8}	Logical OR of AL and imm8 operands, result in AL	1
OR (E)AX, imm	0D {16, 32}	Logical OR of (E)AX and imm operands, result in (E)AX	1

OR performs a logical inclusive-OR on each bit of the two operands. The result is stored in the first operand.

In inclusive-OR operations, a 1 bit is written when either corresponding bits in the operands are 1, otherwise 0 is written. The instruction is used for setting specific bits in a number. For example, ORing the binary value 1000 0000 with any number will set its most-significant (sign) bit.

The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:

AF	undefined
CF	0
OF	0
PF	0 if odd parity, 1 if even parity
SF	high-order bit of result
ZF	0 if result was nonzero, 1 if result was zero

OUT—Output to I/O Port

Instruction	Opcode	Action	Clocks (by mode): rm, vm	pm
OUT imm8, AL	E6	Output byte in AL to port specified by imm8	11	43
OUT imm8, (E)AX	E7	Output word/dword in (E)AX to port specified by imm8	11	43
OUT DX, AL	EE	Output byte in AL to port specified by DX	11	43
OUT DX, (E)AX	EF	Load word/dword in (E)AX from port specified by DX	11	43

OUT copies data from the second operand, a register, and transfers it to the first operand, an I/O port. The second operand is a data byte, word, or dword stored in the AL, AX, or EAX register. The port address is specified either as an 8-bit immediate operand, which can address up to 256 ports, or in the DX register, which can address the full 64kB range of ports.

Table A-18 shows which privilege-level checks are performed against the I/O protection level (IOPL) and the I/O permission bitmap (IOPB) for each mode.

Table A-18. *OUT Privilege Level Checks*

	Protected Mode	Real Mode	Virtual-8086 Mode
IOPL	yes	yes	no
IOPB	yes (for 32-bit tasks)	no	yes

For I/O to succeed, the following must be true: In protected mode for 32-bit tasks, CPL must be ≤ IOPL, or the IOPB bit for the port must be cleared to 0. For 16-bit (80286) tasks, CPL must be ≤ IOPL, and the IOPB is not checked. In real mode, CPL must be ≤ IOPL, but since CPL is always 0, I/O always succeeds. In virtual-8086 mode, IOPL is never checked, and only the IOPB bit for the port must be cleared to 0.

For details, see the sections entitled "Protection Mechanisms" and "Other Processing Modes" in Chapter 4.

Also see the OUTS, OUTSB, OUTSD, and OUTSW instructions for string outputs.

Flags Changed: None

OUTS, OUTSB, OUTSW, and OUTSD—Output to an I/O Port From a String Element

Instruction	Opcode	Action	Clocks (by mode):	rm, vm	pm
OUTSB	6E	Copy byte at address in DS:[(E)SI] to port specified by DX		16	47
OUTSW	6F	Copy word at address in DS:[(E)SI] to port specified by DX		16	47
OUTSD	6F	Copy dword at address in DS:[(E)SI] to port specified by DX		16	47

OUTS copies data from a memory string, specified indirectly as the memory address contained in the DS:(E)SI register, and transfers it to an I/O port, specified in the DX register. A segment override prefix can be used to specify a source location other than the DS segment.

If the DF flag is cleared to 0, the source register is incremented by 1, 2, or 4 (depending on operand size) to point to the next string element; if DF is set to 1, the source register is decremented. The LOOP instruction or the REP instruction prefix can be used to repeat the operation.

In protected mode, the CPL must be less than or equal to the IOPL, and the IOPB bit for the port must be cleared to 0. In real mode, these I/O protections do not apply.

For details on privilege-level checking, see the description of the OUT instruction.

Flags Changed: None

POP—Pop Operand From Stack

Instruction	Opcode	Action	Clocks
POP mem	8F /0	Pop value on top of stack into mem	10
POP reg	58+reg	Pop value on top of stack into reg	2

This form of POP removes the word or dword at the top of the stack and stores it in the register or memory location specified by the instruction's operand. The stack pointer, SS:(E)SP, is then incremented by 2 or 4, depending on operand size, to point to the new top of stack.

Flags Changed: None

POP—Pop Selector Into Segment Register From Stack

Instruction	Opcode	Action	Clocks (by mode):	rm, vm	pm
POP DS	1F	Pop value on top of stack into DS		8	24
POP ES	07	Pop value on top of stack into ES		8	24
POP SS	17	Pop value on top of stack into SS		8	24
POP FS	0F A1	Pop value on top of stack into FS		8*	24*
POP GS	0F A9	Pop value on top of stack into GS		8*	24*

This form of POP removes the word or dword (depending on operand size) at the top of the stack and stores it in the specified segment register. If the item popped is a dword, its upper word is disregarded; selector registers are only 16 bits wide.

The operation initializes the register with the selector value. In protected mode, the operation also causes the processor to load the segment descriptor referenced by the selector into the selector's shadow register. The stack pointer, SS:(E)SP, is then incremented by 2 or 4, depending on operand size, to point to the new top of stack.

After stack segment loads, hardware interrupts (including NMI) are inhibited during the next instruction. This allows the next instruction to load the ESP register with the stack pointer. The sequence should therefore be:

```
POP SS
POP Top_of_stack_pointer
```

For data segment loads (except stack segments), a general-protection fault (exception 13) is generated if the descriptor's DPL is less than the maximum of CPL and the selector's RPL. For stack segment loads, a stack fault (exception 12) is generated if the descriptor's DPL, the selector's RPL, and the CPL are not all equal. The DS and ES registers can be loaded with a null selector. An access to a segment with a null selector will generate a general-protection fault (exception 13).

An invalid-opcode fault (exception 6) is generated if an attempt is made to load the CS segment register, since this would result in a CS value that is unrelated to its associated EIP value. For code segment changes, the CS and EIP registers must be loaded simultaneously. This is done with a far jump or call, return from call, interrupt or exception, or task switch.

For details on privilege-level checking, see the sections entitled "Protection Mechanisms" and "Other Processing Modes" in Chapter 4.

Flags Changed: None

POPA and POPAD—Pop Into All General Registers From Stack

Instruction	Opcode	Action	Clocks
POPAD	61	Pop all general registers (dwords)	19*
POPA	61	Pop all general registers (words)	19*

These instructions remove all eight words (POPA) or dwords (POPAD) at the top of the stack and store them in the general registers. By the end of the operation, the stack pointer, SS:(E)SP, has been incremented by 16 or 32 to point to the new top of stack.

The order of removal and storing is:

(E)DI
(E)SI
(E)BP
(E)SP ← The stack pointer value is discarded
(E)BX
(E)DX
(E)CX
(E)AX

The operation reverses the action of PUSHA and PUSHAD.

Flags Changed: None

POPF and POPFD—Pop (E)FLAGS From Stack

Instruction	Opcode	Action	Clocks
POPFD	9D	Pop dword from stack into EFLAGS register	8*
POPF	9D	Pop word from stack into FLAGS register	8*

POPFD removes the dword from the top of the stack and stores its low-order word in the low-order word of the EFLAGS register. POPF does the same, except that a word is removed from the top of the stack. Neither instruction updates the RF or VM bits in the upper word of EFLAGS.

The stack pointer, SS:(E)SP, is then incremented by 2 or 4, depending on operand size, to point to the new top of stack. The IOPL field is only copied if CPL = 0. The IF field is updated only if CPL ≤ IOPL.

Flags Changed:

CF	restored
PF	restored
AF	restored
ZF	restored
SF	restored
TF	restored
IF	restored only if CPL ≤ IOPL
DF	restored
OF	restored
IOPL	restored only if CPL = 0
NT	restored
RF	not restored
VM	not restored

PUSH—Push Operand Onto Stack

Instruction	Opcode	Action	Clocks
PUSH r/m	FF /6	Push r/m onto stack	6
PUSH reg	50+reg	Push reg onto stack	2
PUSH imm	68	Push imm value onto stack	3
PUSH imm8	6A	Push imm8 value onto stack	4
PUSH DS	1E	Push DS onto stack	3
PUSH ES	06	Push ES onto stack	3
PUSH CS	0E	Push CS onto stack	3
PUSH SS	16	Push SS onto stack	3
PUSH FS	0F A0	Push FS onto stack	3*
PUSH GS	0F A8	Push GS onto stack	3*

PUSH copies the operand—a general register, segment register, memory location, or immediate byte—onto the top of the stack. The operation begins by decrementing the stack pointer by 2 or 4 (depending on operand size). A word or dword, depending on the operand-size attribute, is then pushed on the top of the stack.

If the destination is a segment register and a dword is pushed, the upper word is undefined. If the operand is an 8-bit immediate, it is sign-extended to a word or dword (depending on operand size).

In an instruction like PUSH (E)SP, the pushed value of (E)SP is the value prior to the decrementing of (E)SP that takes place as part of the PUSH operation. This differs from the convention on the 8086 processor.

Flags Changed: None

PUSHA and PUSHAD—Push All General Register Contents

Instruction	Opcode	Action	Clocks
PUSHA	60	Push all general registers onto stack (words)	21*
PUSHAD	60	Push all general registers onto stack (dwords)	21*

These instructions copy all eight word (PUSHA) or dword (PUSHAD) general registers onto the top of the stack. By the end of the operation, the stack pointer, SS:(E)SP, has been decremented by 16 or 32 to point to the new top of stack.

The order of copying is:

(E)AX
(E)CX
(E)DX
(E)BX
(E)SP ← The value at the beginning of the instruction
(E)BP
(E)SI
(E)DI

Flags Changed: None

PUSHF and PUSHFD—Push the Flags Register

Instruction	Opcode	Action	Clocks
PUSHF	9C	Push FLAGS onto the stack	3*
PUSHFD	9C	Push EFLAGS onto the stack	3*

PUSHFD copies the complete dword EFLAGS register onto the top of the stack. PUSHF copies only the low-order word (the FLAGS register).

Flags Changed: None

RCL, RCR, ROL, and ROR—Rotate Left/Right

Instruction	Opcode	Action	Clocks
RCL r/m, CL	D2 /2 {8}, D3 /2 {16, 32}	Rotate r/m and CF left CL times	6/10
RCL r/m, imm8	C0 /2 {8}, C1 /2 {16, 32}	Rotate r/m and CF left imm8 times	6/10
RCL r/m, 1	D0 /2 {8}, D1 /2 {16, 32}	Rotate r/m and CF left once	6/10
RCR r/m, CL	D2 /3 {8}, D3 /3 {16, 32}	Rotate r/m and CF right CL times	6/10
RCR r/m, imm8	C0 /3 {8}, C1 /3 {16, 32}	Rotate r/m and CF right imm8 times	6/10
RCR r/m, 1	D0 /3 {8}, D1 /3 {16, 32}	Rotate r/m and CF right once	6/10
ROL r/m, CL	D2 /0 {8}, D3 /0 {16, 32}	Rotate r/m left CL times	1/5
ROL r/m, imm8	C0 /0 {8}, C1 /0 {16, 32}	Rotate r/m left imm8 times	1/5
ROL r/m, 1	D0 /0 {8}, D1 /0 {16, 32}	Rotate r/m left once	1/5
ROR r/m, CL	D2 /1 {8}, D3 /1 {16, 32}	Rotate r/m right CL times	1/5
ROR r/m, imm8	C0 /1 {8}, C1 /1 {16, 32}	Rotate r/m right imm8 times	1/5
ROR r/m, 1	D0 /1 {8}, D1 /1 {16, 32}	Rotate r/m right once	1/5

These instructions move the bits of the first operand left or right, bit-by-bit, and store the result in the same operand. At one end of the operand, the bits wrap around to the opposite end of the operand. The second operand indicates how many bit movements to perform. Only the low-order five bits of the second operand (32 rotates) are significant.

There are two basic groups of rotate instructions:

- Rotate Through Carry Flag (RLR and RCR)—The bit rotation goes through the carry flag, using it as an additional bit in the rotation sequence, before wrapping around to the other end of the operand.
- Simple Rotate (ROL and ROR)—The bit rotation does not go through the carry flag, but the carry flag is given a copy of the bit value that wraps around to the other end of the operand.

For example, RCR performs a right rotation, through carry. It moves bits toward the least-significant position and shifts the lowest bit (bit 0) to the most-significant position. The left rotation does the opposite. The CF flag is included in the rotations performed by the RCR and RCL instructions. RCR shifts the CF flag into the most-significant bit position, and the least-significant bit is shifted into the CF flag. RCL does the reverse.

After a one-bit rotation, if the high-order bit of the destination operand does not match the carry flag, the OF flag is set to 1. For rotations greater than one bit, the OF flag is undefined.

Flags Changed:	CF	assigned according to the shift
	OF	1 if mismatch with CF after one-bit shift, otherwise undefined

REP, REPE, REPZ, REPNE, and REPNZ—Repeat the String Operation (Instruction Prefix)

Instruction	Opcode	Action	Clocks
REPE CMPSx	F3 A6 {8}, F3 A7 {16, 32}	Compare strings until difference found	4+10n
REPNE CMPSx	F2 A6 {8}, F2 A7 {16, 2}	Compare strings until like elements found	4+10n
REP INSx	F3 6C {8}, F3 6D {16, 32}	Input multiple bytes/words/dwords from port	7+9n
REP LODSx	F2 AC {8}, F2 AD {16, 32}	Copy multiple bytes/words/dwords to AL/AX/EAX	5+6n
REP MOVSx	F3 A4 {8}, F3 A5 {16, 32}	Copy multiple bytes/words/dwords between strings	19+4n
REP OUTSx	F3 6E {8}, F3 6F {16, 32}	Output multiple bytes/words/dwords to port	8+7n
REPE SCASx	F3 AE {8}, F3 AF {16, 32}	Search string until difference found from AL/AX/EAX	5+8n
REPNE SCASx	F2 AE {8}, F2 AF {16, 32}	Search string until element in AL/AX/EAX found	5+8n
REP STOSx	F3 AA {8}, F3 AB {16, 32}	Fill memory region with value in AL/AX/EAX	5+6n

The REP instruction prefix and its variants repeat the instruction they precede, decrementing the count value in the (E)CX register until (E)CX = 0. The distinctions between the prefixes are shown in Table A-19.

Table A-19. *REP Prefixes*

Instruction	Termination Condition
REP	(E)CX = 0
REPE	(E)CX = 0, or ZF = 1
REPNE	(E)CX = 0, or ZF = 0

In the clock counts, n refers to the number of iterations in the repeating operation.

Flags Changed:

AF	depends on instruction being prefixed
CF	depends on instruction being prefixed
OF	depends on instruction being prefixed
SF	depends on instruction being prefixed
PF	depends on instruction being prefixed
ZF	depends on instruction being prefixed

RET (near)—Return to Calling Procedure in Same Segment

Instruction	Opcode	Action	Clocks
RET/RETN imm16	C2	Near return, pop number of bytes specified by imm16	10
RET/RETN	C3	Near return	10

The near (intra-segment) return instruction passes control from a called procedure back to the calling procedure within the same segment. Typically, the called procedure was accessed with a CALL instruction. Upon return, execution continues at the instruction following the CALL instruction.

The RET and RETN mnemonics are synonyms. The RET mnemonic, which refers to either a near return or a far return, is interpreted properly by assemblers.

The instruction pops the (E)IP from the top of the stack and branches to that address, which is the instruction following the original CALL instruction. RET/RETN imm16 pops and discards imm16 parameter bytes after the return address is popped and before branching, to remove the parameters pushed onto the stack by the caller. The operand size of items popped from the stack depends on the operand-size attribute.

Flags Changed: None

RET (far)—Return to Calling Procedure in Different Segment

Instruction	Opcode	Action	Clocks (by mode):	rm, vm	pm
RET/RETF imm16	CA	Far return, pop number of bytes specified by imm16		17*	See Table A-20
RET/RETF	CB	Far return		17*	See Table A-20

The far (inter-segment) return instruction passes control from a called procedure back to the calling procedure in a different code segment. Typically, the called procedure is accessed with a CALL instruction. Upon return, execution continues at the instruction following the CALL instruction.

In protected mode and virtual-8086 mode, the number of clocks required depends on the destination of the return, as shown in Table A-20.

Table A-20. *RET Clock Counts*

Privilege Level	RETF imm16	RETF
To same privilege level	55	53
To inner (more privileged) level	149	146

RET and RETF mnemonics are synonyms. The RET mnemonic, which refers to either a near return or a far return, is interpreted properly by assemblers.

The instruction first pops the (E)IP, then a CS selector from the top of the stack. RETF imm16 also pops and discards imm16 parameter bytes. In real and virtual-8086 modes, the program then branches to the address and code segment that were popped.

In protected mode, the code segment descriptor's access rights and the code segment selector's RPL are checked before branching to the popped address and segment. If the return is to a more privileged level (lower RPL value for the destination code-segment selector), the stack of the called procedure will have the caller's original (E)SP as its last entry. This entry is popped so that a stack switch can be made before execution resumes in the calling procedure. See the sections entitled "Segmentation," "Protection Mechanisms," and "Other Processing Modes" in Chapter 4 for details on segment descriptor access rights and privilege-level checking.

The operand size of items popped from the stack depends on the operand-size attribute. In protected-mode far returns, the operand size must match the size of the call gate that accessed the procedure. If the procedure was accessed without a gate—either through a conforming-segment call or a call at the same privilege level—the operand size of the CALL instruction must match that of the RETF instruction.

Flags Changed: None

SAHF—Store AH Register Into EFLAGS

Instruction	Opcode	Action	Clocks
SAHF	9E	Copy AH into FLAGS register	3*

SAHF copies the contents of the AH register into the low-order byte of the EFLAGS register. This byte contains all arithmetic flags except OF.

Flags Changed:		
	OF	unchanged
	SF	AH bit 7
	ZF	AH bit 6
	AF	AH bit 4
	PF	AH bit 2
	CF	AH bit 0

SAL, SAR, SHL, and SHR—Shift Arithmetic Left/Right

Instruction	Opcode	Action	Clocks
SAL/SHL r/m, CL	D2 /4 {8}, D3 /4 {16, 32}	Shift r/m left CL times	1/5
SAL/SHL r/m, imm8	C0 /4 {8}, C1 /4 {16, 32}	Shift r/m left imm8 times	1/5
SAL/SHL r/m, 1	D0 /4 {8}, D1 /4 {16, 32}	Shift r/m left once	1/5
SAR r/m, CL	D2 /7 {8}, D3 /7 {16, 32}	Shift arithmetic r/m right CL times	1/5
SAR r/m, imm8	C0 /7 {8}, C1 /7 {16, 32}	Shift arithmetic r/m right imm8 times	1/5
SAR r/m, 1	D0 /7 {8}, D1 /7 {16, 32}	Shift arithmetic r/m right once	1/5
SHR r/m, CL	D2 /5 {8}, D3 /5 {16, 32}	Shift r/m right CL times	1/5
SHR r/m, imm8	C0 /5 {8}, C1 /5 {16, 32}	Shift r/m right imm8 times	1/5
SHR r/m, 1	D0 /5 {8}, D1 /5 {16, 32}	Shift r/m right once	1/5

These instructions shift the bits of the first operand left or right, bit-by-bit, and store the result in the same operand. At one end of the operand, the shifted bits are discarded; at the other end they are filled with zeros. The second operand indicates how many bit shifts to perform. Only the low-order five bits of this operand (indicating 32 bit shifts) are significant.

There are two basic groups of shift instructions:

- Shift Right (SAR and SHR)
- Shift Left (SAL/SHL)

The SAR and SHR instructions shift bits to the right, effectively dividing the operand by 2 with each shift. The least-significant bit that is shifted out is copied to the carry flag. The two instructions differ as follows:

SAR fills vacated high-order bits with the original sign bit. This results in a signed two's-complement divide with rounding toward negative infinity. The instruction works differently than the IDIV instruction for negative numbers. When SAR gets to -1, it cannot divide the number further, as IDIV can.

SHR fills vacated high-order bits with zeros and clears the sign bit. This results in an unsigned two's-complement divide and works like the DIV instruction.

SAL and SHL are synonyms. These instruction shift bits to the left, effectively multiplying the operand by 2 with each shift. The vacated low-order bits are filled with zeros. The most-significant bit that is shifted out is copied to the carry flag. SAL/SHL work like the MUL instruction, except when the result does not fit in the same operand size as the original multiplicands.

The OF flag is assigned the values shown in Table A-21.

Table A-21. *OF Flag Values*

Instruction	One-bit shifts	Multi-bit shifts
SAL/SHL	1	undefined
SAR	0	undefined
SHR	msb[1]	undefined

[1] msb—most significant bit of the first operand before the shift.

Flags Changed:

AF	undefined
CF	SAL/SHL—high-order bit shifted out; SAR/SHR—low-order bit shifted out
OF	see Table A-21.
PF	0 if odd parity, 1 if even parity
SF	high-order bit of result
ZF	0 if result was nonzero, 1 if result was zero

SBB—Subtract With Borrow

Instruction	Opcode	Action	Clocks
SBB r, r/m	1A {8}, 1B {16, 32}	Subtract CF + r/m operand from r	1/5
SBB r/m, r	18 {8}, 19 {16, 32}	Subtract CF + r operand from r/m	1/5
SBB r/m, imm	80 /3 {8}, 81 /3 {16, 32}	Subtract CF + imm operand from same-size r/m	1/5
SBB r/m, imm8	83 /3 {16, 32}	Subtract CF + imm8 operand from r/m	1/5
SBB AL, imm8	1C	Subtract CF + imm8 operand from AL	1
SBB (E)AX, imm	1D {16, 32}	Subtract CF + imm operand from (E)AX	1

SBB adds the CF flag to the source operand, subtracts that value from the destination operand, and stores the result in the destination operand. The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:

AF	0 if no borrow to low nibble, 1 if borrow
CF	0 if no borrow to high-order bit, 1 if borrow
OF	0 if no overflow, 1 if overflow
PF	0 if odd parity, 1 if even parity
SF	high-order bit of result
ZF	0 if result was nonzero, 1 if result was zero

SCALL—Call SuperState V

Instruction	Opcode	Action	Clocks
SCALL r/m	0F 18	Invoke the SuperState V function indicate by r/m	21-103

SCALL is used to invoke either a SuperState V hardware function or a SuperState V program. The hardware functions provide basic control over SuperState V enabling and over the Super386 processor's cache. SuperState V programs can be written to manage power, virtual I/O, and other SuperState V functions independently of the operating system and application programs that are running.

If a hardware function succeeds, a return parameter may be written to the operand, and the carry flag is cleared to 0. If the function fails, an error code of -1 is written to the operand and the carry flag is set to 1.

The vectors for the functions shown in Table A-22.

Table A-22. *SCALL Vector Functions*

Vector	Clocks	Description
0	21	CPU Version—The processor returns, in the operand, a 32-bit code that is divided into two 8-bit fields indicating the processor type and processor stepping level, as follows: **Bits** **Meaning** 31:16 Reserved 15:8 Processor stepping level 7:0 Processor type: 0 = 38600DXE 1 = 38600SXE 2 = 38605DXE 3 = 38605SXE
1	71	Enable SuperState V Mode Using Primary Descriptor—Enables the SuperState V operating mode, using the primary segment descriptor at address 000FFFC0. SCALL functions entering SuperState V mode must not be executed until SuperState V mode is enabled. For proper operation, SuperState V mode should first be initialized and then enabled using this function. The CPL must be 0, and the code and data required for this call must be initialized prior to the call. If the required code and data are not initialized prior to the call, the processor will probably crash.

Table A-22. SCALL Vector Functions (continued)

Vector	Clocks	Description
2	78	Enable SuperState V Mode Using Secondary Descriptor—Enables the SuperState V operating mode using the secondary segment descriptor at address 000EFFC0. Otherwise, same as function 1.
3	97/100	Execute the SuperState V Program—For any vector with bit 31 set to 1, SuperState V mode will be entered (if enabled) at the offset indicated by the SCALL entry vector. Vectors with bit 31 cleared to 0 are reserved. Before executing the instructions following the SCALL instruction, the processor automatically (a) copies the SCALL vector into the EDX register, (b) copies the instruction's MODr/m byte into the BH register, and (c) stores the contents of the EIP, EFLAGS, EDX, EBX, and CS registers into the SuperState V save area in memory. The saved EIP point to the instruction following the SCALL.
4	33	Disable Cache—Disables the instruction cache, if CPL = 0.
5	37	Enable Cache—Enables the instruction cache, if CPL = 0.
6	35	Query Cache—Writes the following result to the operand: I = Instruction cache is enabled. 0 = Instruction cache is disabled.
7	84	Flush Cache—Flushes the contents of the instruction cache.
80000000 to FFFFFFFF	97/100	Execute The SuperState V Program—For any vector with bit 31 set to 1, SuperState V mode will be entered (if enabled) at the offset indicated by the SCALL entry vector. Vectors with bit 31 cleared to 0 are reserved. Before executing the instructions following the SCALL instruction, the processor automatically (a) copies the SCALL vector into the EDX register, (b) copies the instruction's MODr/m byte into the BH register, and (c) stores the contents of the EIP, EFLAGS, EDX, EBX, and CS registers into the SuperState V save area in memory. The saved EIP point to the instruction following the SCALL.

The SCALL instruction is not included in the standard 80386 instruction set and will generate an invalid opcode fault (exception 6) on processors other than the Super386 processor.

Flags Changed: CF = 0 if function succeeds, 1 if it fails.

SCASB, SCASW, and SCASD—Scan String Data

Instruction	Opcode	Action	Clocks
SCASB	AE	Scan string at address in ES:[(E)DI] for AL	8
SCASW	AF	Scan string at address in ES:[(E)DI] for AX	8
SCASD	AF	Scan string at address in ES:[(E)DI] for EAX	8

These instructions compare the contents of the AL, AX, or EAX register with the memory location (byte, word, or dword) addressed indirectly by the ES:(E)DI register. The flags are set in the same manner as the CMPB, CMPW, and CMPD instructions.

If the DF flag is cleared to 0, the memory address in the destination register, ES:(E)DI, is incremented by 1, 2, or 4 (depending on operand size) to point to the next string element. If DF is set to 1, the register is decremented. The LOOP instruction or the REP instruction prefix can be used to repeat the operation.

The result of the comparison, which is done by subtraction, is discarded. The address-size attribute determines whether the ES:DI or ES:EDI register stores the memory location. The ES segment referenced by the (E)DI offset cannot be overridden with an instruction prefix.

See the CMPB, CMPW, and CMPD instructions.

Flags Changed:

AF	0 if no borrow to low nibble, 1 if borrow
CF	0 if no borrow to high-order bit, 1 if borrow
OF	0 if no overflow, 1 if overflow
PF	0 if odd parity, 1 if even parity
SF	high-order bit of result
ZF	0 if result was nonzero, 1 if result was zero

SETcc—Set Byte on Condition

Instruction	Opcode	Action	Clocks
SETE/SETZ r/m8	0F 94	r/m8 = 1 if ZF = 1 otherwise r/m8 = 0	2*/7*
SETNE/SETNZ r/m8	0F 95	r/m8 = 1 if ZF = 0 otherwise r/m8 = 0	2*/7*
SETA/SETNBE r/m8	0F 97	r/m8 = 1 if CF = 0 and ZF = 0 otherwise r/m8 = 0	2*/7*
SETBE/SETNA r/m8	0F 96	r/m8 = 1 if CF = 1 or ZF = 1 otherwise r/m8 = 0	2*/7*
SETB/SETNAE r/m8	0F 92	r/m8 = 1 if CF = 1 otherwise r/m8 = 0	2*/7*
SETAE/SETNB r/m8	0F 93	r/m8 = 1 if CF = 0 otherwise r/m8 = 0	2*/7*
SETG/SETNLE r/m8	0F 9F	r/m8 = 1 if ZF = 0 or SF = OF otherwise r/m8 = 0	2*/7*
SETGE/SETNL r/m8	0F 9D	r/m8 = 1 if SF = OF otherwise r/m8 = 0	2*/7*
SETL/SETNGE r/m8	0F 9C	r/m8 = 1 if SF <> OF otherwise r/m8 = 0	2*/7*
SETLE/SETNG r/m8	0F 9E	r/m8 = 1 if ZF = 1 or SF <> OF otherwise r/m8 = 0	2*/7*
SETS r/m8	0F 98	r/m8 = 1 if SF = 1 otherwise r/m8 = 0	2*/7*
SETNS r/m8	0F 99	r/m8 = 1 if SF = 0 otherwise r/m8 = 0	2*/7*
SETO r/m8	0F 90	r/m8 = 1 if OF = 1 otherwise r/m8 = 0	2*/7*
SETNO r/m8	0F 91	r/m8 = 1 if OF = 0 otherwise r/m8 = 0	2*/7*
SETP r/m8	0F 9A	r/m8 = 1 if PF = 1 otherwise r/m8 = 0	2*/7*
SETNP r/m8	0F 9B	r/m8 = 1 if PF = 0 otherwise r/m8 = 0	2*/7*

These instructions set the byte operand to 1 if the condition listed above is true. If the condition is false, the byte is cleared to 0. Some opcodes have more than one mnemonic, because their effects can be interpreted in different ways. In the mnemonics listed above, the following abbreviations are used:

A	above (for comparing unsigned integers)
B	below (for comparing unsigned integers)
C	carry
E	equal to
G	greater than (for comparing signed integers)
L	less than (for comparing signed integers)
N	not
O	overflow
P	parity
S	sign
Z	zero

Flags Changed: None

SGDT—Store Global Descriptor Table Register

Instruction	Opcode	Action	Clocks
SGDT m	0F 01 /0	Copy global descriptor table register to m	10*

SGDT copies the contents of the GDTR to a six-byte memory structure that is addressed by the destination operand.

For 32-bit operands, a two-dword memory structure is used. The first dword begins with a word for the segment limit, followed by the low-order word of the segment base. The second dword contains the high-order word of the segment base. The upper word of the second dword is undefined.

For 16-bit operands, a three-word memory structure is used. The first word is the segment limit. The second word is the low-order word of the segment base. The first byte of the third word is the high-order byte of the segment base. The upper byte of the third word is undefined.

See the section entitled "Descriptor Tables and Their Registers" in Chapter 4 for details.

Unlike the LGDT instruction, SGDT can be executed from any privilege level.

Flags Changed: None

SHLD—Shift Left Double

Instruction	Opcode	Action	Clocks
SHLD r/m, r, CL	0F A5	Shift r/m + r left CL times; result in r/m	6*/9*
SHLD r/m, r, imm8	0F A4	Shift r/m + r left imm8 times; result in r/m	6*/9*

SHLD shifts the bits of the first operand left and stores the result in the second operand. The vacated low-order bits are filled with the high-order bits of the second operand. The third operand indicates how many bit shifts to perform; only the low-order five bits of this operand (indicating 32 bit shifts) are significant.

The high-order bit shifted out of the first operand is copied to the CF flag. The OF flag is set to 1 if the most-significant bit of the first operand (the sign bit of the result) after the shift does not match the carry flag; otherwise, OF is cleared to 0.

Flags Changed:

AF	undefined
CF	low-order bit shifted out
OF	undefined
PF	0 if odd parity, 1 if even parity
SF	high-order bit of result
ZF	0 if result was nonzero, 1 if result was zero

SHRD—Shift Right Double Precision

Instruction	Opcode	Action	Clocks
SHRD r/m, r, CL	0F AD	Shift r/m + r right CL times; result in r/m	6*/9*
SHRD r/m, r, imm8	0F AC	Shift r/m + r right imm8 times; result in r/m	6*/9*

SHRD shifts the bits of the first operand right and stores the result in the second operand. The vacated high-order bits are filled with the low-order bits of the second operand. The third operand indicates how many bit shifts to perform; only the low-order five bits of this operand (indicating 32 bit shifts) are significant.

The low-order bit shifted out of the first operand is copied to the CF flag.

Flags Changed:

AF	undefined
CF	low-order bit shifted out
OF	undefined
PF	0 if odd parity, 1 if even parity
SF	high-order bit of result
ZF	0 if result was nonzero, 1 if result was zero

SIDT—Store Interrupt Descriptor Table Register

Instruction	Opcode	Action	Clocks
SIDT r/m16	0F 01 /1	Copy interrupt descriptor table register to r/m16	10*

SIDT copies the contents of the IDTR to a six-byte memory structure addressed by the destination operand.

For 32-bit operands, a two-dword memory structure is used. The first dword begins with a word for the segment limit, followed by the low-order word of the segment base. The second dword contains the high-order word of the segment base. The upper word of the second dword is undefined.

For 16-bit operands, a three-word memory structure is used. The first word is the segment limit. The second word is the low-order word of the segment base. The first byte of the third word is the high-order byte of the segment base. The upper byte of the third word is undefined.

See the section entitled "Descriptor Tables and Their Registers" in Chapter 4 for details.

Unlike the LIDT instruction, SIDT is not a privileged instruction and can be executed from any privilege level.

Flags Changed: None

SLDT—Store Local Descriptor Table Register

Instruction	Opcode	Action	Clocks
SLDT r/m16	0F 00 /0	Copy local descriptor table register to r/m16	4*/5*

SLDT copies the 16-bit contents of the LDTR to the operand.

LDTs are only used in protected mode. Like the LLDT instruction, SLDT can only be used from privilege level 0.

See the section entitled "Descriptor Tables and Their Registers" in Chapter 4 for details.

Flags Changed: None

SMSW—Store Machine Status Word

Instruction	Opcode	Action	Clocks
SMSW r/m16	0F 01 /4	Copy machine status word to r/m16	3*/4*

SMSW is provided for compatibility with the 80286. It is not recommended for use in new Super386 code. Use the MOV instruction instead. SMSW copies the lower word of control register CR0, called the machine status word (MSW), into the instruction's operand.

Unlike the LMSW instruction, SMSW can be used from any privilege level.

Flags Changed: None

STC—Set Carry Flag

Instruction	Opcode	Action	Clocks
STC	F9	Set CF = 1	2

STC sets the carry flag (CF) to 1.

Flags Changed: CF = 1

STD—Set Direction Flag

Instruction	Opcode	Action	Clocks
STD	FD	Set DF = 1	2

STD sets the direction flag (DF) to 1. Following an STD instruction, string instructions decrement their index registers (E)SI and/or (E)DI. The DF settings are:

DF = 1 Decrement (E)SI and (E)DI
DF = 0 Increment (E)SI and (E)DI

Flags Changed: DF = 1

STI—Set Interrupt Flag

Instruction	Opcode	Action	Clocks
STI	FB	Set IF = 1	5

STI sets the interrupt flag (IF) to 1. When STI is executed, the processor will respond to external interrupts after the instruction following STI has completed, and until the IF flag is cleared to 0.

In protected mode, the CPL must be less than or equal to IOPL.

The flag is cleared with the CLI instruction.

Flags Changed: IF = 1

STOSB, STOSW, and STOSD—Load String Operands

Instruction	Opcode	Action	Clocks
STOSB	AA	Store byte in AL at address in ES:[(E)DI]	5
STOSW	AB	Store word in AX at address in ES:[(E)DI]	5
STOSD	AB	Store dword in EAX at address in ES:[(E)DI]	5

STOSB, STOSW, and STOSD copy the contents (byte, word, or dword) of the AL, AX, or EAX register to the memory location addressed indirectly by the ES:(E)DI register.

If the DF flag is cleared to 0, the destination register is incremented by 1, 2, or 4 (depending on operand size) to point to the next string element. If DF is set to 1, the destination register is decremented. The LOOP instruction or the REP instruction prefix can be used to repeat the operation.

Offset (E)DI is referenced to the ES segment register and cannot be overridden with an instruction prefix.

Flags Changed: None

STR—Store Task Register

Instruction	Opcode	Action	Clocks
STR r/m16	0F 00 /1	Copy task register contents to r/m16	4*/5*

STR copies the task register, which contains the selector for current TSS, into the operand. The instruction operates only in protected mode.

Flags Changed: None

SUB—Integer Subtraction

Instruction	Opcode	Action	Clocks
SUB r, r/m	2A {8}, 2B {16, 32}	Subtract r/m operand from r	1/5
SUB r/m, r	28 {8}, 29 {16, 32}	Subtract r operand from r/m	1/5
SUB r/m, imm	80 /5 {8}, 81 /5 {16, 32}	Subtract imm operand from same-size r/m	1/5
SUB r/m, imm8	83 /5 {16, 32}	Subtract imm8 operand from r/m	1/5
SUB AL, imm8	2C	Subtract imm8 operand from AL	1
SUB AX/EAX, imm	2D {16, 32}	Subtract imm operand from AX/EAX	1

SUB subtracts the second operand from the first operand and stores the result in the first operand. The instruction operates on signed or unsigned integers.

The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:

AF 0 if no borrow to low nibble, 1 if borrow
CF 0 if no borrow to high-order bit, 1 if borrow
OF 0 if no overflow, 1 if overflow
PF 0 if odd parity, 1 if even parity
SF high-order bit of result
ZF 0 if result was nonzero, 1 if result was zero

TEST—Logical Bit Test

Instruction	Opcode	Action	Clocks
TEST r/m, r	84 {8}, 85 {16, 32}	Logical AND of r/m and r operands	2/6
EST r/m, imm	F6 /0 {8}, F7 /0 {16, 32}	Logical AND of r/m and imm operands	1/5
TEST AL, imm8	A8	Logical AND of r/m and imm8 operands	2
TEST (E)AX, imm	A9 {16, 32}	Logical AND of (E)AX and imm operands	2

TEST does a logical AND of the two operands. The result is discarded but the arithmetic flags (except AF) are valid.

The instruction can be used, for example, to determine if either operand is nonzero (ZF = 0). Unlike the AND instruction, TEST does not alter the first operand.

Flags Changed:

AF undefined
CF 0
OF 0
PF 0 if odd parity, 1 if even parity
SF high-order bit of result
ZF 0 if result was nonzero, 1 if result was zero

VERR and VERW—Verify Segment for Read/Write

Instruction	Opcode	Action	Clocks
VERR r/m16	0F 00 /4	ZF = 1 if segment indicated by r/m16 is readable otherwise ZF = 0	23*/26*
VERW r/m16	0F 00 /5	ZF = 1 if segment indicated by r/m16 is writable otherwise ZF = 0	23*/26*

These instructions determine whether the segment referenced by the selector in the operand is one of the following:

- Defined (within the limits of the GDT or an LDT)
- A code or data segment (not a TSS, gate, or descriptor table)
- Readable (VERR) or writable (VERW)
- Reachable according to the architecture's privilege-level rules.

These instructions cannot be used in real or virtual-8086 mode.

For details on privilege-level checking, see the sections entitled "Protection Mechanisms" and "Other Processing Modes" in Chapter 4.

Flags Changed: ZF = 1 if all conditions are met; otherwise 0

WAIT—Wait Until Not Busy

Instruction	Opcode	Action	Clocks
WAIT	9B	Wait for BUSY input signal to go inactive	2*

WAIT is designed for synchronizing processor and coprocessor interactions. It idles the processor until the BUSY signal goes inactive, indicating that the coprocessor is able to accept another command from the processor. The BUSY signal can be asserted by other devices if the system does not have a coprocessor installed, enabling the WAIT instruction to halt execution until the signal is deasserted.

WAIT should be issued before accessing data stored by the coprocessor, and at the end of any program that uses the coprocessor. The mnemonics WAIT and FWAIT are equivalent.

Flags Changed: None

XCHG—Exchange Register With Memory or Register

Instruction	Opcode	Action	Clocks
XCHG r, r/m	86 {8}, 87 {16, 32}	Exchange r and r/m values	4/7
XCHG (E)AX, reg	90+reg	Exchange (E)AX and reg values	3*

XCHG exchanges the values in the first and second operands. The operands may be in any order. If one operand is a memory operand, the LOCK signal is asserted during the instruction operation. The LOCK prefix has no effect on this instruction.

Flags Changed: None

XLATB—Translate Byte via Table Lookup

Instruction	Opcode	Action	Clocks
XLATB	D7	Copy byte at DS:(E)BX+AL into AL	5*

XLATB uses AL as an offset into a table in memory whose base is pointed to by DS:(E)BX. The referenced entry (a byte) is copied into AL, overwriting the original offset.

The address-size attribute determines whether EBX or BX points to the base of the table. A segment-override instruction prefix can be used to reference a segment other than DS.

Flags Changed: None

XOR—Bitwise Exclusive-OR

Instruction	Opcode	Action	Clocks
XOR r, r/m	32 {8}, 33 {16,32}	XOR of r/m and r operands, result in r	1/5
XOR r/m, r	30 {8}, 31 {16,32}	XOR of r/m and r operands, result in r/m	1/5
XOR r/m, imm	80 /6 {8}, 81 /6 {16, 32}	XOR of r/m and imm operands, result in r/m	1/5
XOR r/m, imm8	83 /6 {16, 32}	XOR of r/m and imm8 operands, result in r/m	1/5
XOR AL, imm8	34	XOR of AL and imm8 operands, result in AL	1
XOR (E)AX, imm	35 {16, 32}	XOR of (E)AX and imm operands, result in (E)AX	1

XOR performs a logical exclusive-OR on each bit of the two operands. The result is stored in the first operand.

In exclusive-OR operations, a 1 bit is written when the corresponding bits in the operands consist of a 1 and a 0. If there are two 1s or two 0s, a 0 is written. The instruction is useful for setting specific bits in a number. For example, XORing a value with itself clears the value to 0.

The LOCK prefix can be used with this instruction when a memory operand is modified as a result of the operation.

Flags Changed:

AF	undefined
CF	0
OF	0
PF	0 if odd parity, 1 if even parity
SF	high-order bit of result
ZF	0 if result was nonzero, 1 if result was zero

APPENDIX B

Super386 Quick Reference

This appendix summarizes the features of the Super386 microprocessor in the following sections:

- System Register Reference
- Protected Mode Reference
- Instruction Reference
- Address Mode Reference
- Opcodes.

System Register Reference

Figure B-1 provides an overview of the Super386 registers. It includes the instruction pointer, flag, general purpose, and segment registers.

Figure B-1. *System Register Overview*

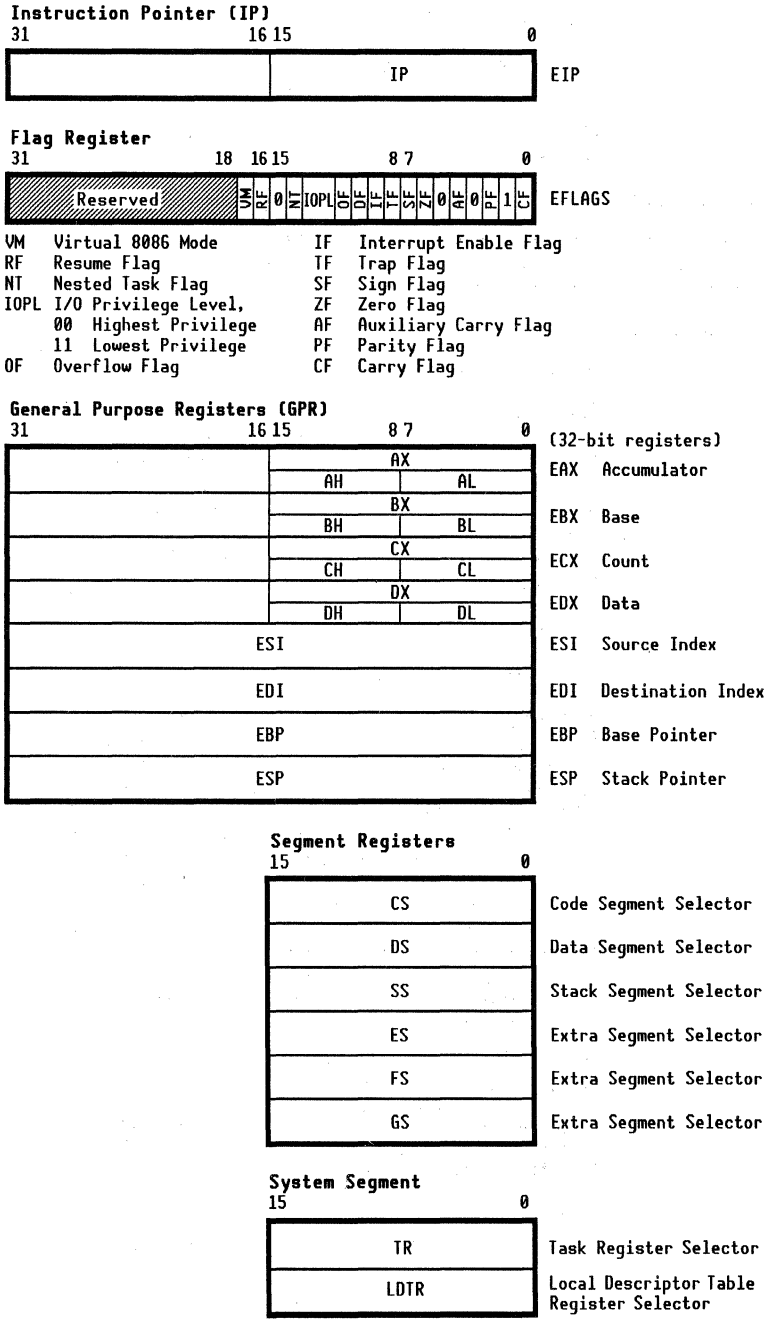
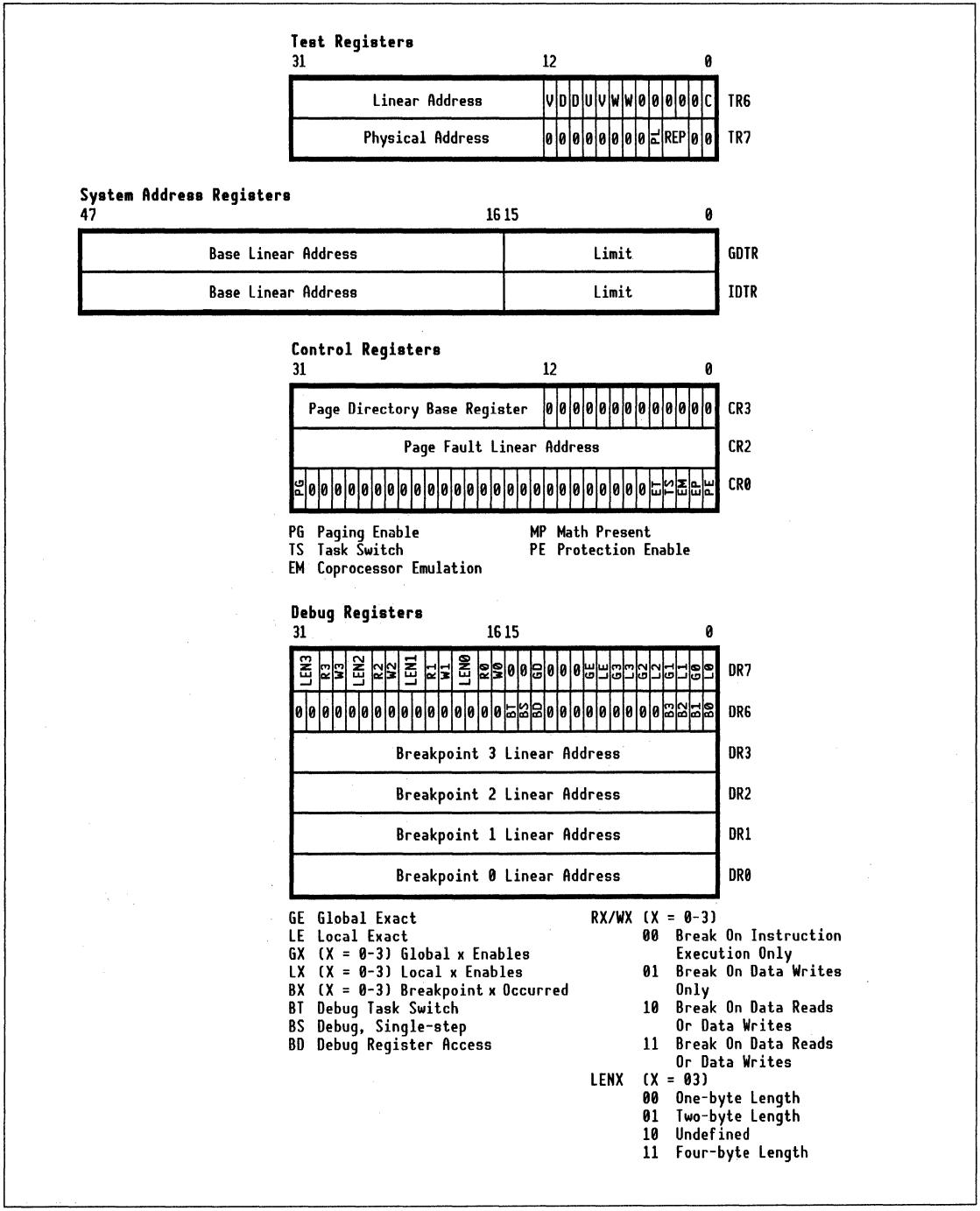


Figure B-1. *System Register Overview (continued)*



Protected Mode Reference

Figure B-2 shows the selector register format and the segment registers. The selectors point to the descriptors in the global descriptor table (GDTR) or the local descriptor table (LDTR) as specified by the TI bit of the selector.

In Figures B-3 through B-6, the 16-bit descriptors specify the format for a 286 descriptor. The 32-bit descriptors specify the format for a Super386 descriptor.

Figure B-2. *Selector Register and Shadow*

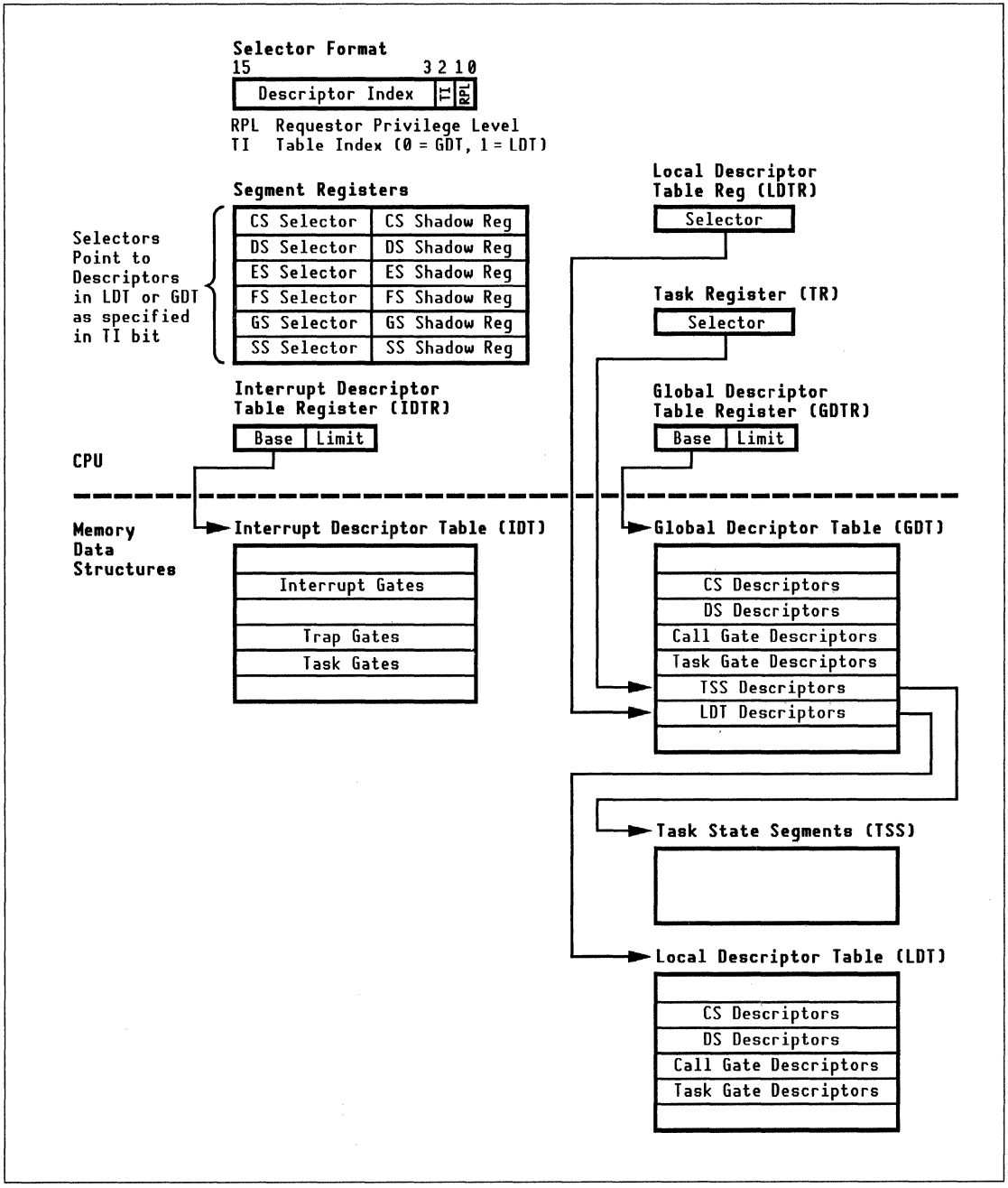


Figure B-3. *Non-System Segment Descriptors*

Code Segment Descriptor (Super386 32-bit Type)

31 30 29 28 27 26 25 24	23	22	21	20	19 18 17 16	15	14 13	12	11	10	9	8	7 6 5 4 3 2 1 0	
Base 31:24	G	D	0	AVL	Limit 19:16	P	DPL	1	1	C	R	A	Base 23:16	+4
Segment Base 15:0						Segment Limit 15:0								+0

Code Segment Descriptor (286 16-bit Type)

31 30 29 28 27 26 25 24	23	22	21	20	19 18 17 16	15	14 13	12	11	10	9	8	7 6 5 4 3 2 1 0	
0 0 0 0 0 0 0 0	0	0	0	0	0 0 0 0	P	DPL	1	1	C	R	A	Segment Base 23:16	+4
Segment Base 15:0						Segment Limit 15:0								+0

Data Segment Descriptor (Super386 32-bit Type)

31 30 29 28 27 26 25 24	23	22	21	20	19 18 17 16	15	14 13	12	11	10	9	8	7 6 5 4 3 2 1 0	
Segment Base 31:24	G	B	0	AVL	Limit 19:16	P	DPL	1	0	E	W	A	Segment Base 23:16	+4
Segment Base 15:0						Segment Limit 15:0								+0

Data Segment Descriptor (286 16-bit Type)

31 30 29 28 27 26 25 24	23	22	21	20	19 18 17 16	15	14 13	12	11	10	9	8	7 6 5 4 3 2 1 0	
0 0 0 0 0 0 0 0	0	0	0	0	0 0 0 0	P	DPL	1	0	E	W	A	Base 23:16	+4
Segment Base 15:0						Segment Limit 15:0								+0

Key:

A Accessed
AVL Available to Software
B Big (see Table B-1)
C Conforming
D Default Operand and Address Size (16-bit or 32-bit)
DPL Descriptor Privilege Level
E Expand Down (see Table B-1)
G Granularity
P Segment Present
R Read Enable
W Write Enable

Table B-1 describes the E-bit and B-bit encodings in the data segment descriptor.

Table B-1. *E-bit and B-bit Encoding*

E Bit	B Bit	Descriptor's Use	Resulting Segment Characteristics
0	X[1]	In non-stack data segment: DS, ES, FS, or GS	Expand-up data segment
0	0	In stack data segment: SS	Expand-up 16-bit stack segment[2]
0	1	In SS	Expand-up 32-bit stack segment[3]
1	0	In DS, ES, FS, or GS	Expand-down data segment, upper limit[4] = (64k - 1)
1	0	In SS	Expand-down 16-bit stack segment[2], upper limit[4] = (64k - 1)
1	1	In DS, ES, FS, or GS	Expand-down data segment, upper limit[4] = (4G - 1)
1	1	In SS	Expand-down 32-bit stack segment[3], upper limit[4] = (4G - 1)

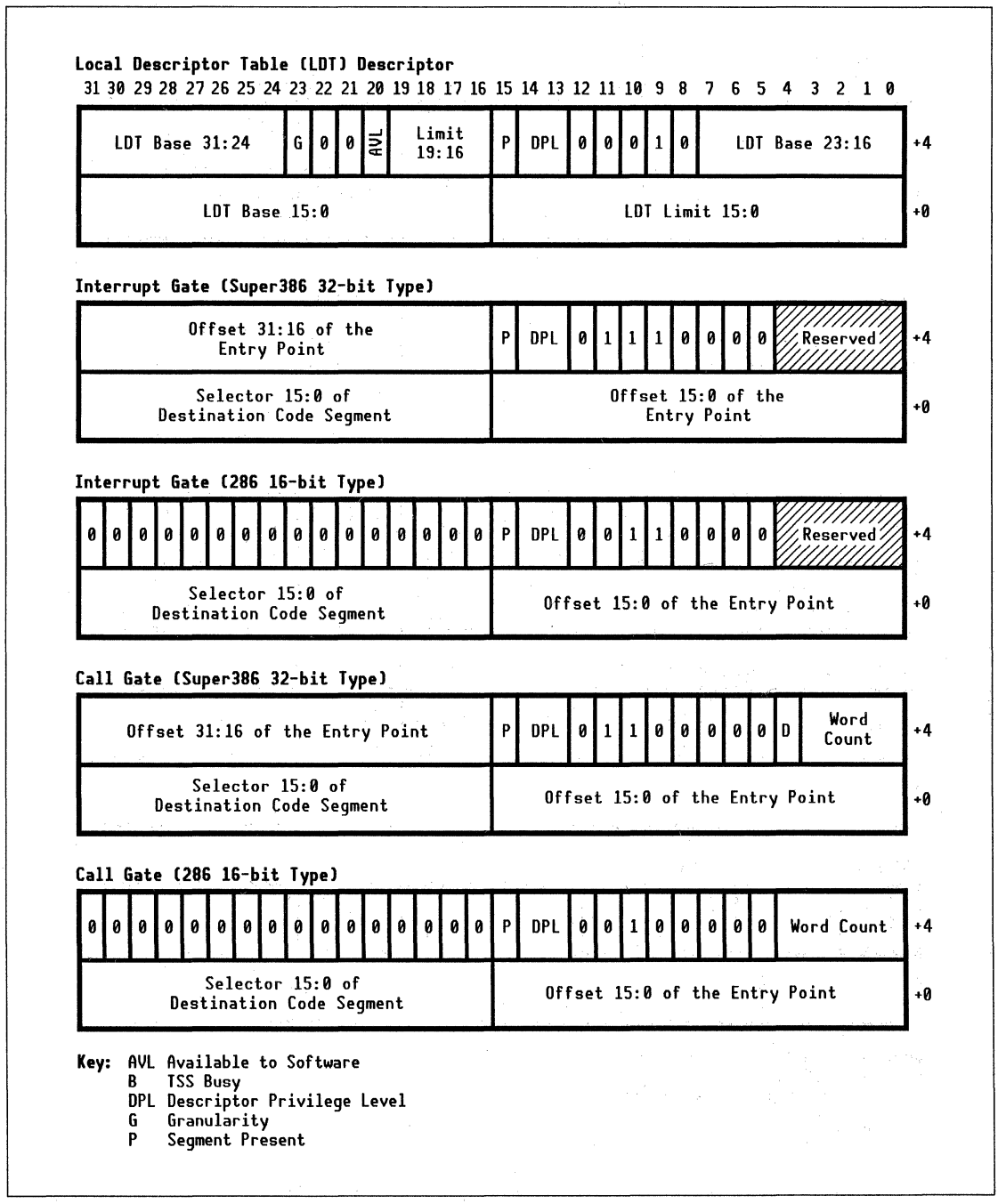
1 Value of this bit is 0 or 1.

2 Implicit stack references are 16-bit and SP register is updated.

3 Implicit stack references are 32-bit and ESP register is updated.

4 Valid offsets in an expand-down segment are between an upper limit, which is spcified by the B bit, and the segment limit, which is defined by the segment descriptor's segment limit field and G bit.

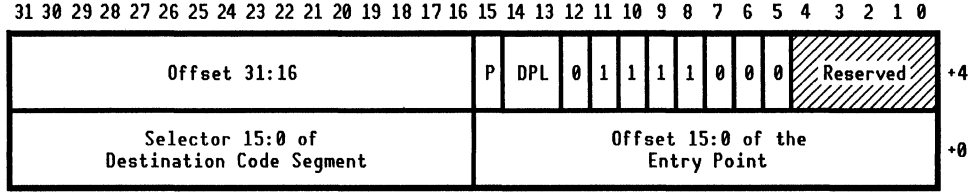
Figure B-4. *System Segment Descriptors*



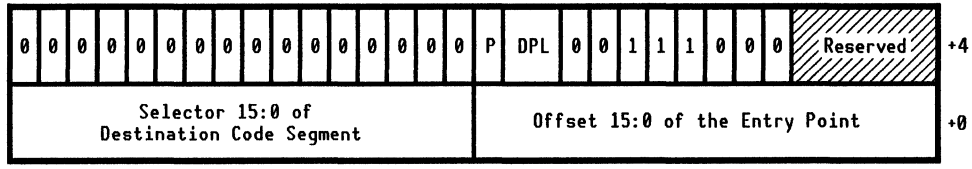
Key: AVL Available to Software
B TSS Busy
DPL Descriptor Privilege Level
G Granularity
P Segment Present

Figure B-4. *System Segment Descriptors (continued)*

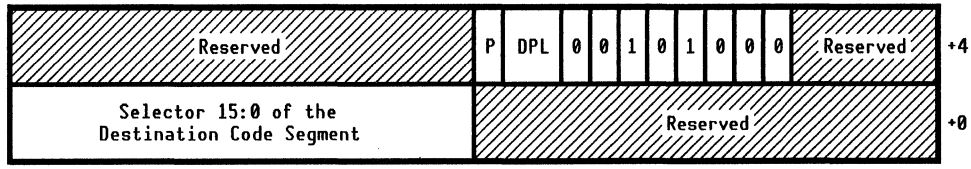
Trap Gate (Super386 32-bit Type)



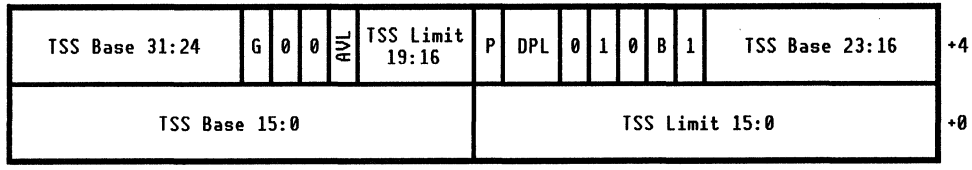
Trap Gate (286 16-bit Type)



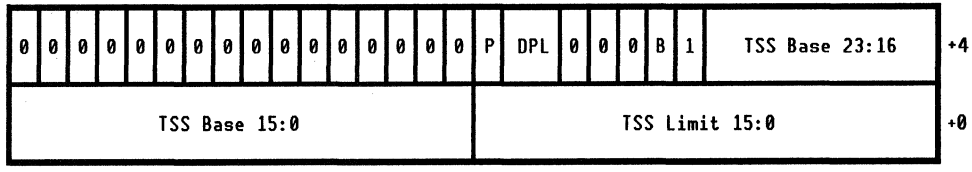
Task Gate



TSS Descriptor (Super386 32-bit Type)



TSS Descriptor (286 16-bit Type)



Key: AVL Available to Software
B TSS Busy
DPL Descriptor Privilege Level
G Granularity
P Segment Present

Figure B-5. Super386 Task State Segment (TSS) Structure

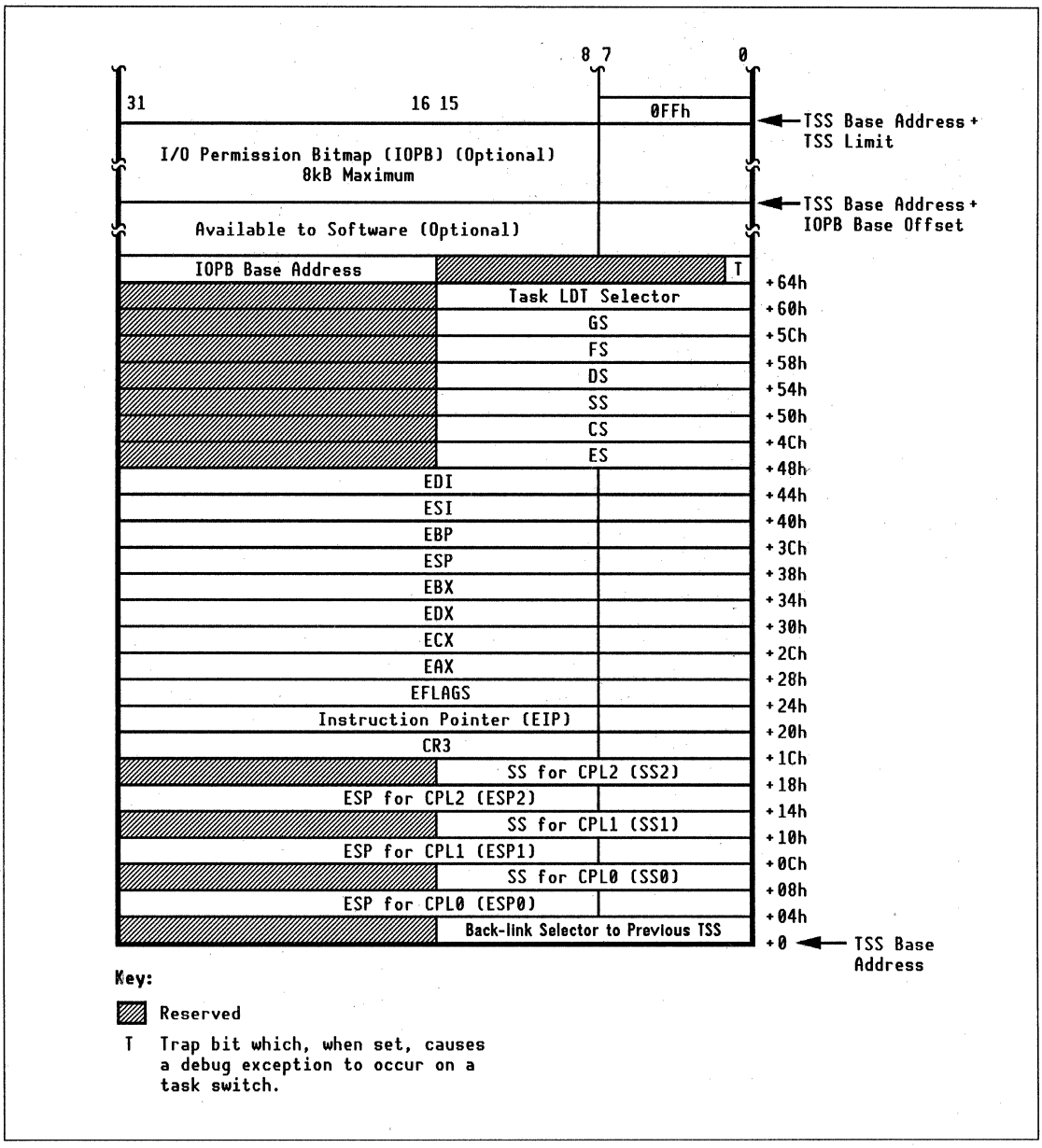
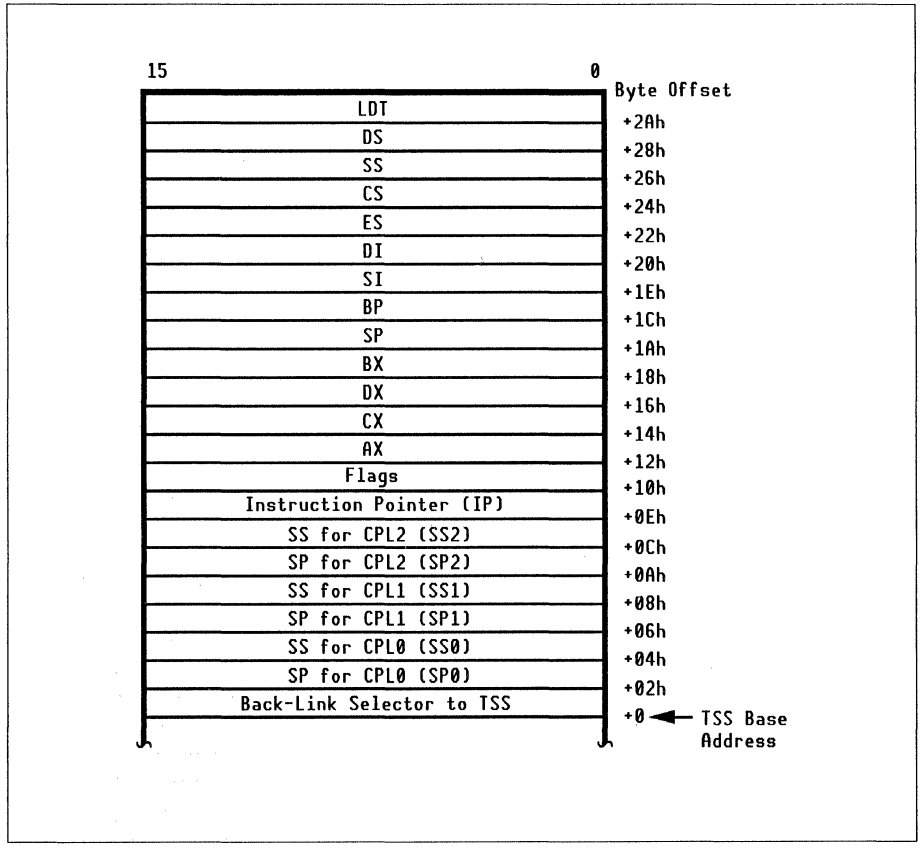


Figure B-6. *80286 Task State Segment (TSS) Structure*



Instruction Reference

Table B-2 is a summary of the Super386 instruction set. It presents the instruction set in quick reference format to show the modified flags, modified locations, and types of exceptions each instruction may encounter. Opcodes that have multiple encodings share common entries in the table. ADD, for example, shows both a register and a memory location being updated. Any single ADD operation can only modify one of these operands.

Descriptions of the Table B-2 entries and definitions of the Flags, Registers, Memory, Exceptions, and Other column headings are provided in Tables B-3 through B-7.

Table B-2. *Super386 Instruction Summary*

	Flags (See Table B-3)													Regs (See	Memory (See	Exceptions	Other (See
Instruction	V	R	N	IO	O	D	I	T	S	Z	A	P	C	Table B-4)	Table B-5)	(See Table B-6)	Table B-7)
AAA					u				u	u	M	u	M	AX			
AAD					u				M	M	u	M	u	AX			I
AAM					u				M	M	u	M	u	AX		D0	I
AAS					u				u	u	M	u	M	AX			
ADC					M				M	M	M	M	M	GPR	MODr/m	MEM	I
ADD					M				M	M	M	M	M	GPR	MODr/m	MEM	I
AND					M				M	M	u	M	M	GPR	MODr/m	MEM	I
ARPL										M				GPR	MODr/m	R6, MEM	
BOUND[1]																I5, M6, MEM	B, T
BSF					u				u	M	u	u	u	GPR		MEM	
BSR					u				u	M	u	u	u	GPR		MEM	
BT					u				u	u	u	u	M	GPR	MODr/m	MEM	I
BTC					u				u	u	u	u	M	GPR	MODr/m	MEM	I
BTR					u				u	u	u	u	M	GPR	MODr/m	MEM	I
BTS					u				u	u	u	u	M	GPR	MODr/m	MEM	I
CALL[1]														ESP	STACK	T13, T10, MEM	T
CBW														AX			
CDQ														EDX, EAX			
CLC													M				
CLD						M											
CLI							M									P13	
CLTS																C13	
CMC													M				
CMP					M				M	M	M	M	M	GPR	MODr/m	MEM	I

1 The flags are modified during these instruction only if a task switch occurs. If a task switch does not occur, the flags are affected only as noted.

Table B-2. *Super386 Instruction Summary (continued)*

Instruction	Flags (See Table B-3)													Regs (See Table B-4)	Memory (See Table B-5)	Exceptions (See Table B-6)	Other (See Table B-7)
	V	R	N	IO	O	D	I	T	S	Z	A	P	C				
CMPS					M				M	M	M	M	M	ESI, EDI		MEM	B
CWD														DX, AX			
CWDE														EAX			
DAA					u				M	M	M	M	M	AL			
DAS					u				M	M	M	M	M	AL			
DEC					M				M	M	M	M		GPR	MODr/m	MEM	
DIV					u				u	u	u	u	u	EDX, EAX		D0, MEM	
ENTER														ESP, EBP	STACK	MEM	B
HLT																C13	
IDIV					u				u	u	u	u	u	EDX, EAX		D0, MEM	
IMUL					M				u	u	u	u	M	GPR		MEM	I
IN														EAX		TSS	I
INC					M				M	M	M	M		GPR	MODr/m	MEM	
INS														EDI	STRING	TSS, MEM	
INT[1]			M					M								T13, T10, MEM	I, T
INTO[1]			M					M								T13, T10, MEM	T
IRET[2]	M	M	M	M	M	M	M	M	M	M	M	M	M			T13, T10, MEM	B, T
Jcond																J13	T
JCXZ																J13	T
JMP[1]																T13, T10, MEM, J13	T
LAHF														AH			
LAR										M				GPR		R6, MEM	
LDS														DS, GPR		D13, M6, N11, MEM	B
LEA														GPR		M6	
LES														ES, GPR		D13, M6, N11, MEM	B
LEAVE														ESP, EPB		MEM	
LFS														FS, GPR		D13, M6, N11, MEM	B
LGDT														GDTR		C6, MEM, M6	B
LGS														GS, GPR		D13, M6, N11, MEM	B
LIDT														IDTR		C6, M6, MEM	B

1 The flags are modified during these instruction only if a task switch occurs. If a task switch does not occur, the flags are affected only as noted.

2 During an IRET, the V and R flags are not modified. The R-flag is modified during an IRETD. The V-flag is modified during an IRETD when in protected mode and the CPL equals zero. The I/O flags are modified during IRET or IRETD when the CPL equals zero. The I-flag is modified when the current I/O Privilege Level (IOPL) is of equal or lesser privilege than the CPL privilege. The remaining flags are always modified during an IRET or IRETD.

Table B-2. *Super386 Instruction Summary (continued)*

Instruction	Flags (See Table B-3)													Regs (See Table B-4)	Memory (See Table B-5)	Exceptions (See Table B-6)	Other (See Table B-7)
	V	R	N	IO	O	D	I	T	S	Z	A	P	C				
LLDT														LDTR		R6, C6, MEM, N11	B
LMSW														CR0		MEM	
LOCK																L6	
LODS														ESI, EAX		MEM	
LOOP														ECX		J13	T
LSL										M				GPR		R6, MEM	
LSS														SS, GPR		S13, M6, N11, MEM	B
LTR														TR		R6, C6, MEM, N11	B
MOV														SEL *or* GPR	MODr/m	MEM	I
MOV CR					u				u	u	u	u	u	CR *or* GPR		C6, G6	
MOV DR					u				u	u	u	u	u	DR *or* GPR		C6, G6	
MOV TR					u				u	u	u	u	u	TR *or* GPR		C6, G6	
MOVS														ESI , EDI	STRING	MEM	B
MOVSX														GPR		MEM	
MOVZX														GPR		MEM	
MUL					M				u	u	u	u	M	EDX, EAX		MEM	
NEG					M				M	M	M	M	M	GPR	MODr/m	MEM	
NOP																	
NOT														GPR	MODr/m	MEM	
OR					M				M	M	u	M	M	GPR	MODr/m	MEM	I
OUT															PORT	TSS	I
OUTS														ESI	PORT	TSS, MEM	
POP														SEL *or* GPR		MEM, N11	
POPA														all GPR		MEM	B
POPF[3]			M	M	M	M	M	M	M	M	M	M	M	ESP		I13, MEM	
PUSH														ESP	STACK	MEM	
PUSHA														ESP	STACK	MEM	B
PUSHF														ESP	STACK	I13, MEM	
RCL					W								M	GPR	MODr/m	MEM	I
RCR					W								M	GPR	MODr/m	MEM	I
REP														ECX			B
RET[1]														ESP		T13, MEM	I, T

1 The flags are modified during these instruction only if a task switch occurs. If a task switch does not occur, the flags are affected only as noted.

3 POPF or POPFD never modify the V and R flags. They only modify the I/O flags if the CPL equals zero, and only modify the I-flag if the current IOPL is of equal or lesser privilege than the CPL privilege. The remaining flags are always modified during a POPF or POPFD.

Table B-2. *Super386 Instruction Summary (continued)*

Instruction	Flags (See Table B-3)													Regs (See Table B-4)	Memory (See Table B-5)	Exceptions (See Table B-6)	Other (See Table B-7)
	V	R	N	IO	O	D	I	T	S	Z	A	P	C				
ROL					W								M	GPR	MODr/m	MEM	I
ROR					W								M	GPR	MODr/m	MEM	I
SAHF									M	M	M	M	M				
SAL					W				M	M	u	M	M	GPR	MODr/m	MEM	I
SAR					W				M	M	u	M	M	GPR	MODr/m	MEM	I
SBB					M				M	M	M	M	M	GPR	MODr/m	MEM	I
SCAS					M				M	M	M	M	M	EDI		MEM	
SETcond														GPR	MODr/m	MEM	
SGDT															MODr/m	M6, MEM	B
SHL					W				M	M	u	M	M	GPR	MODr/m	MEM	I
SHR					W				M	M	u	M	M	GPR	MODr/m	MEM	I
SHLD					u				M	M	u	M	M	GPR	MODr/m	MEM	I
SHRD					u				M	M	u	M	M	GPR	MODr/m	MEM	I
SIDT															MODr/m	M6, MEM	B
SLDT														GPR	MODr/m	R6, MEM	
SMSW														GPR	MODr/m	MEM	
STC													M				
STD						M											
STI							M									P13	
STOS														EDI	STRING	MEM	
STR														GPR	MODr/m	R6, MEM	
SUB					M				M	M	M	M	M	GPR	MODr/m	MEM	I
TEST					M				M	M	u	M	M			MEM	I
VERR										M						R6, MEM	
VERW										M						R6, MEM	
WAIT																NPX	
XCHG														GPR	MODr/m	MEM	B
XLAT														AL		MEM	
XOR					M				M	M	u	M	M	GPR	MODr/m	MEM	I
SCALL													M	GPR		new, MEM	

Table B-3 defines the codes used in the Flags column of Table B-2.

Table B-3. *Super386 Instruction Summary—Flags Description*

Flag	Description	Bits	Flag	Description	Bits
V	Virtual-8086 flag	17	S	Sign flag	7
R	Resume flag	16	Z	Zero flag	6
N	Nested task flag	14	A	Auxiliary carry flag	4
IO	I/O privilege level	13:12	P	Parity flag	2
O	Overflow flag	11	C	Carry flag	0
D	Direction flag	10	M	Flag is modified	
I	Interrupt flag	9	u	Flag is undefined	
T	Trap flag	8	W	Modified if rotate/shift amount is 1; otherwise undefined	

Table B-4 defines the codes used in the Registers (Regs) column of Table B-2.

Table B-4. *Super386 Instruction Summary—Registers Description*

Name	Description	Name	Description
GPR	One of eight GPRs is modified	SEL *or* GPR	Selector or GPR is modified
SEL	One of six selectors is modified	CR *or* GPR	Control register or GPR is modified
CR	One of three control registers is modified	DR *or* GPR	Debug register or GPR is modified
DR	One of six debug registers is modified	TR *or* GPR	Test register or GPR is modified
TR	One of two test registers is modified		

Table B-5 defines the codes used in the Memory column of Table B-2.

Table B-5. *Super386 Instruction Summary—Memory Description*

Type	Memory Location
MODr/m	Memory or register location pointed to by the MODr/m encoding is modified
STACK	Memory location pointed to by SS:(E)SP is modified
STRING	Memory location pointed to by ES:(E)DI is modified
PORT	Output port location pointed to by DX is modified

Table B-6 defines the codes used in the Exceptions column of Table B-2.

Table B-6 *Super386 Instruction Summary—Exceptions Description*

Exception	Description
C6	In protected mode, the instruction causes a general protection exception (13) if the CPL does not equal zero.
C13	In protected mode, the instruction causes a general protection exception (13) if the CPL is not equal to zero. The instruction causes a general protection exception (13) in virtual-8086 mode.
D0	Instruction will encounter a divide exception if the denominator is zero, or if the result is too large to fit into the destination operand (0).
D13	In protected mode, a non-readable or data/nonconforming code segment where a requested privilege level (RPL) or CPL has less privilege than the DPL will signal a general protection exception (13).
G6	The instruction's MOD field of the MODr/m byte must indicate a register operand; otherwise, an undefined opcode exception (6) is signaled.
I5	Instruction causes an interrupt 5 if the register operand does not lie between the memory operands (5). Instruction is invalid if the MOD field of the MODr/m byte indicates a register operand (6).
I13	In virtual-8086 mode, an instruction causes a general protection exception (13) if the I/O privilege level does not equal 3.
J13	In protected mode, if the jump target address is beyond the code segment limit, a general protection exception (13) is signaled.
L6	The LOCK prefix can only occur with one of the following instructions; otherwise, an undefined opcode exception (6) is signaled. Instructions that can use the LOCK prefix are ADC, ADD, AND, BTC, BTR, BTS, OR, SBB, SUB, or XOR with the operands (memory, register), XCHG with the operands (memory, register), XCHG with the operands (register, memory), and DEC, INC, NEG, or NOT with the operand (memory).
MEM	Instruction using memory operands can encounter memory operand exceptions under the following conditions: a. When executing in real or virtual-8086 mode, part or all of the operand is not within the effective address space of 0000h to 0FFFFh. In this case, a general protection exception (13) is signaled. b. When executing in protected or virtual-8086 mode with paging enable, the translation mechanism can signal a page fault exception (14). c. In protected mode, an attempt to read or write beyond the segment limit, write a nonwritable data segment, read a nonreadable code sigment, or write to a code segment signals a general protection exception (13). d. When executing in protected mode, an attempt to read or write beyond the segment limit or write a nonwritable stack data segment signals a stack fault exception (12). e. When the operand lies within the LDT, IDT, or GDT, and the operand does not lie within the effective address space of the descriptor table's limit value, a general protection exception (13) is signaled.
M6	The instruction's MOD field of the MODr/m byte must indicate a memory operand; otherwise an undefined opcode exception (6) is signaled.
new	The SCALL instruction acts as a gateway into SuperState V. SuperState V software can reflect exceptions back to the program containing the SCALL if it determines that the operation is invalid or if the requesting program is insufficiently privileged.
NPX	The instruction causes a coprocessor not available exception (7) if the TS-bit and the MP-bit in CR0 are set. A math fault exception (16) occurs if the coprocessor's ERROR pin is asserted.
N11	In protected mode, loading a segment with the Present bit off signals a not-present exception (11) unless the instruction is loading the stack segment. Loading the stack segment with a not-present segment signals a stack fault (12).

Table B-6 *Super386 Instruction Summary—Exceptions Description (continued)*

Exception	Description
P13	In protected mode, an instruction causes a general protection exception (13) if the I/O privilege level has greater privilege than the CPL privilege. The instruction ignores privilege level in real mode.
R6	Instruction is invalid in real and virtual-8086 modes (6).
S13	In protected mode, loading a null-selector, a nondata segment, or a data segment where the DPL does not equal the CPL or the RPL does not equal the CPL signals a general protection exception (13).
TSS	In protected mode, an instruction causes a general protection exception (13) if the I/O privilege level has greater privilege than the CPL privilege or at least one of the corresponding TSS I/O bits is set. The instruction ignores privilege level and permission bits in real mode. In virtual-8086 mode, the instruction causes a general protection exception (13) if at least one of the corresponding TSS I/O bits is set.
T13	During a task switch, loading a null-selector CS, a nonexecutable segment, a conforming CS where the DPL is of less privilege that the CPL, or a non-conforming CS where the DPL does not equal the destination CPL or RPL will signal a general protection exception (13). Having an instruction pointer that does not lie within the effective address space of the CS limit will signal an invalid TSS exception (10).
T10	In protected mode, the instruction that causes a task switch must not encounter any type of fault or exception when accessing the data from the TSS. If a fault or exception would occur, an invalid TSS exception (10) is signaled.

Table B-7 defines the codes used in the Other column of Table B-2.

Table B-7. *Super386 Instruction Summary—Other Description*

Other	Description
I	Instruction encodings are available where one of the source operands is an immediate value contained within the instruction.
B	Instruction requires multiple bus accesses to fetch memory operands.
T	Instruction may alter the normal sequential execution of instructions and cause the instruction buffer to be flushed.

Address Mode Reference

Figures B-7 through B-10 describe Super386 address modes and byte formats. Tables B-8, B-9, and B-10 present the opcodes for the 16-bit address MODr/m, 32-bit address MODr/m, and 32-bit address SIB encodings, respectively.

Figure B-7. *Registers Used in 16-Bit Effective Address Generation*

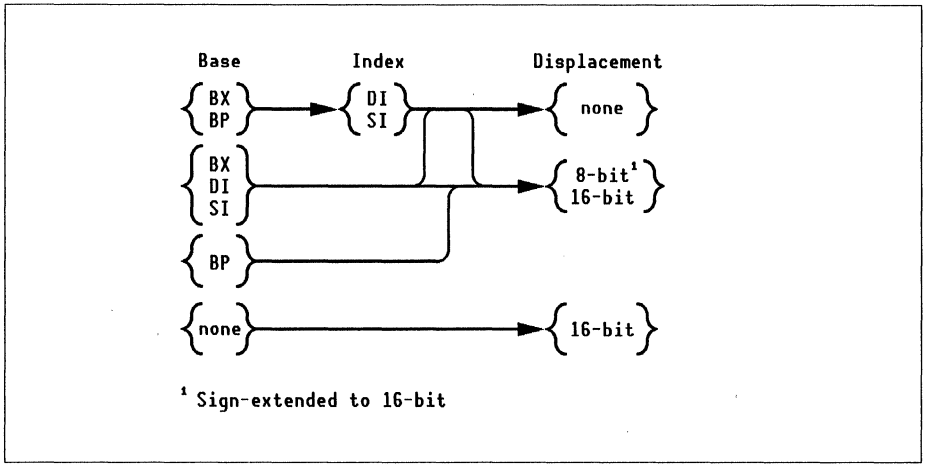


Figure B-8. *Registers Used in 32-Bit Effective Address Generation*

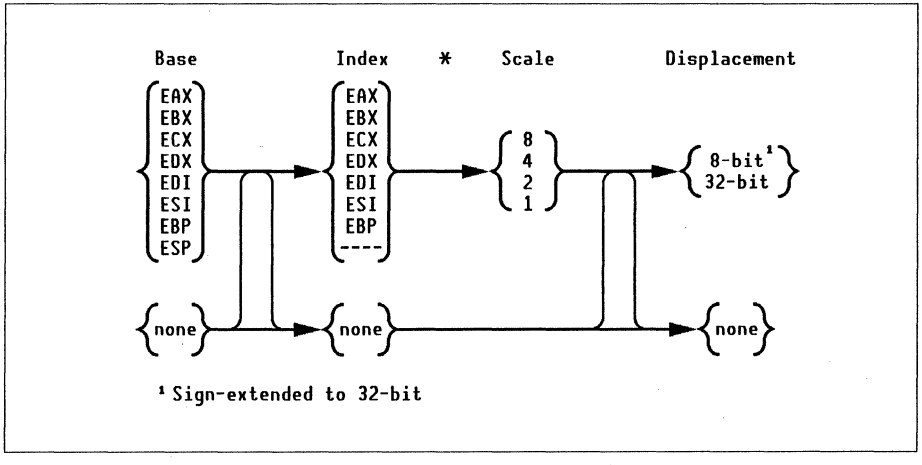


Figure B-9. *MODr/m Byte Format*

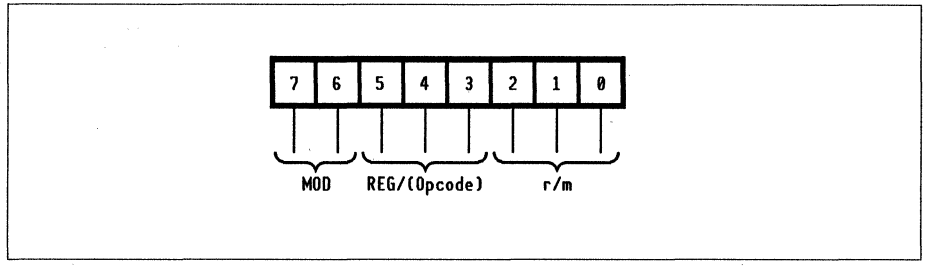


Figure B-10. *SIB Byte Format*

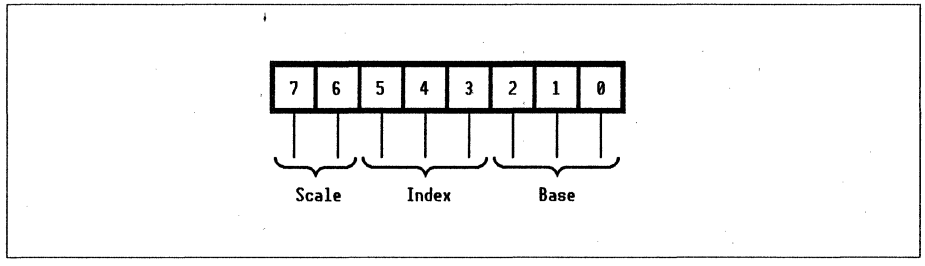


Table B-8. *Super386 16-Bit Address MODr/m Encodings*

MODr/m Byte Format for 16-Bit Addressing Mode			REG (Opcode)							
			000	001	010	011	100	101	110	111
		DWORD[1]	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
		WORD[1]	AX	CX	DX	BX	SP	BP	SI	DI
		BYTE[1]	AL	CL	DL	BL	AH	CH	DH	BH
		GRP1[2]	(ADD)	(OR)	(ADC)	(SBB)	(AND)	(SUB)	(XOR)	(CMP)
		GRP2[2]	(ROL)	(ROR)	(RCL)	(RCR)	(SHL)	(SHR)	(SHL)	(SAR)
		GRP3[2]	(TEST)	(TEST)	(NOT)	(NEG)	(MUL)	(IMUL)	(DIV)	(IDIV)
		GRP4[2]	(INC)	(DEC)	( )	( )	( )	( )	( )	( )
		GRP5[2]	(INC)	(DEC)	(CALL)	(CALL)	(JMP)	(JMP)	(PUSH)	( )
		GRP6[2]	(SLDT)	(STR)	(LLDT)	(LTR)	(VERR)	(VERW)	( )	( )
		GRP7[2]	(SGDT)	(SIDT)	(LGDT)	(LIDT)	(SMSW)	( )	(LMSW)	( )
		GRP8[2]	( )	( )	( )	( )	(BT)	(BTS)	(BTR)	(BTC)
Effective Address	**r/m**	**MOD Field**	**MODr/m Byte Values**							
{BX + SI}	000	00	00h	08h	10h	18h	20h	28h	30h	38h
{BX + SI + disp8}[3]		01	40h	48h	50h	58h	60h	68h	70h	78h
{BX + SI + disp16}[3]		10	80h	88h	90h	98h	A0h	A8h	B0h	B8h
EAX/AX/AL		11	C0h	C8h	D0h	D8h	E0h	E8h	F0h	F8h
{BX + DI}	001	00	01h	09h	11h	19h	21h	29h	31h	39h
{BX + DI + disp8}[3]		01	41h	49h	51h	59h	61h	69h	71h	79h
{BX + DI + disp16}[3]		10	81h	89h	91h	99h	A1h	A9h	B1h	B9h
ECX/CX/CL		11	C1h	C9h	D1h	D9h	E1h	E9h	F1h	F9h
{BP + SI}	010	00	02h	0Ah	12h	1Ah	22h	2Ah	32h	3Ah
{BP + SI + disp8}[3]		01	42h	4Ah	52h	5Ah	62h	6Ah	72h	7Ah
{BP + SI + disp16}[3]		10	82h	8Ah	92h	9Ah	A2h	AAh	B2h	BAh
EDX/DX/DL		11	C2h	CAh	D2h	DAh	E2h	EAh	F2h	FAh
{BP + DI}	011	00	03h	0Bh	13h	1Bh	23h	2Bh	33h	3Bh
{BP + DI + disp8}[3]		01	43h	4Bh	53h	5Bh	63h	6Bh	73h	7Bh
{BP + DI + disp16}[3]		10	83h	8Bh	93h	9Bh	A3h	ABh	B3h	BBh
EBX/BX/BL		11	C3h	CBh	D3h	DBh	E3h	EBh	F3h	FBh
{SI}	100	00	04h	0Ch	14h	1Ch	24h	2Ch	34h	3Ch
{SI + disp8}[3]		01	44h	4Ch	54h	5Ch	64h	6Ch	74h	7Ch
{SI + disp16}[3]		10	84h	8Ch	94h	9Ch	A4h	ACh	B4h	BCh
ESP/SP/AH		11	C4h	CCh	D4h	DCh	E4h	ECh	F4h	FCh
{DI}	101	00	05h	0Dh	15h	1Dh	25h	2Dh	35h	3Dh
{DI + disp8}[3]		01	45h	4Dh	55h	5Dh	65h	6Dh	75h	7Dh
{DI + disp16}[3]		10	85h	8Dh	95h	9Dh	A5h	ADh	B5h	BDh
EBP/BP/CH		11	C5h	CDh	D5h	DCh	E5h	EDh	F5h	FDh
{BP}	110	00	06h	0Eh	16h	1Eh	26h	2Eh	36h	3Eh
{BP + disp8}[3]		01	46h	4Eh	56h	5Eh	66h	6Eh	76h	7Eh
{BP + disp16}[3]		10	86h	8Eh	96h	9Eh	A6h	AEh	B6h	BEh
ESI/SI/DH		11	C6h	CEh	D6h	DEh	E6h	EEh	F6h	FEh
{BX}	111	00	07h	0Fh	17h	1Fh	27h	2Fh	37h	3Fh
{BX + disp8}[3]		01	47h	4Fh	57h	5Fh	67h	6Fh	77h	7Fh
{BX + disp16}[3]		10	87h	8Fh	97h	9Fh	A7h	AFh	B7h	BFh
EDI/DI/BH		11	C7h	CFh	D7h	DFh	E7h	EFh	F7h	FFh

1 When the Super386 opcode indicates a Gb, Gv, or Gw as one of its operands, the register is specified by the REG field of the MODr/m byte and the operand size, e.g. if the operand size is 16-bit, the Word registers are used.

2 When the Super386 opcode indicates a group instruction, the instruction within the group is specified by the REG (opcode) field of the MODr/m byte, e.g., for Super386 opcode F7h having a MODr/m byte with the REG (opcode) field equal to 100 indicates a MUL.

3 disp8/16 indicates that an 8/16-bit, sign-extended displacement follows the MODr/m byte and must be added to the Base and/or Index Value. For effective addresses using the EBP, the default selector is the SS. All other effective addresses use the DS as the default selector.

Table B-9. *Super386 32-Bit Address MODr/m Encodings*

MODr/m Byte Format for 32-Bit Addressing Mode			REG (Opcode)							
			000	001	010	011	100	101	110	111
		DWORD[1]	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
		WORD[1]	AX	CX	DX	BX	SP	BP	SI	DI
		BYTE[1]	AL	CL	DL	BL	AH	CH	DH	BH
		GRP1[2]	(ADD)	(OR)	(ADC)	(SBB)	(AND)	(SUB)	(XOR)	(CMP)
		GRP2[2]	(ROL)	(ROR)	(RCL)	(RCR)	(SHL)	(SHR)	(SHL)	(SAR)
		GRP3[2]	(TEST)	(TEST)	(NOT)	(NEG)	(MUL)	(IMUL)	(DIV)	(IDIV)
		GRP4[2]	(INC)	(DEC)	( )	( )	( )	( )	( )	( )
		GRP5[2]	(INC)	(DEC)	(CALL)	(CALL)	(JMP)	(JMP)	(PUSH)	( )
		GRP6[2]	(SLDT)	(STR)	(LLDT)	(LTR)	(VERR)	(VERW)	( )	( )
		GRP7[2]	(SGDT)	(SIDT)	(LGDT)	(LIDT)	(SMSW)	( )	(LMSW)	( )
		GRP8[2]	( )	( )	( )	( )	(BT)	(BTS)	(BTR)	(BTC)
Effective Address	**r/m**	**MOD Field**	**MODr/m Byte Values**							
{EAX}	000	00	00h	08h	10h	18h	20h	28h	30h	38h
{EAX + disp8}[3]		01	40h	48h	50h	58h	60h	68h	70h	78h
{EAX + disp32}[3]		10	80h	88h	90h	98h	A0h	A8h	B0h	B8h
EAX/AX/AL		11	C0h	C8h	D0h	D8h	E0h	E8h	F0h	F8h
{ECX}	001	00	01h	09h	11h	19h	21h	29h	31h	39h
{ECX + disp8}[3]		01	41h	49h	51h	59h	61h	69h	71h	79h
{ECX + disp32}[3]		10	81h	89h	91h	99h	A1h	A9h	B1h	B9h
ECX/CX/CL		11	C1h	C9h	D1h	D9h	E1h	E9h	F1h	F9h
{EDX}	010	00	02h	0Ah	12h	1Ah	22h	2Ah	32h	3Ah
{EDX + disp8}[3]		01	42h	4Ah	52h	5Ah	62h	6Ah	72h	7Ah
{EDX + disp32}[3]		10	82h	8Ah	92h	9Ah	A2h	AAh	B2h	BAh
EDX/DX/DL		11	C2h	CAh	D2h	DAh	E2h	EAh	F2h	FAh
{EBX}	011	00	03h	0Bh	13h	1Bh	23h	2Bh	33h	3Bh
{EBX + disp8}[3]		01	43h	4Bh	53h	5Bh	63h	6Bh	73h	7Bh
{EBX + disp32}[3]		10	83h	8Bh	93h	9Bh	A3h	ABh	B3h	BBh
EBX/BX/BL		11	C3h	CBh	D3h	DBh	E3h	EBh	F3h	FBh
{— + —}[3]	100	00	04h	0Ch	14h	1Ch	24h	2Ch	34h	3Ch
{— + — + disp8}[3]		01	44h	4Ch	54h	5Ch	64h	6Ch	74h	7Ch
{— + — + disp32}[3]		10	84h	8Ch	94h	9Ch	A4h	ACh	B4h	BCh
ESP/SP/AH		11	C4h	CCh	D4h	DCh	E4h	ECh	F4h	FCh
{disp32}	101	00	05h	0Dh	15h	1Dh	25h	2Dh	35h	3Dh
{EBP + disp8}[3]		01	45h	4Dh	55h	5Dh	65h	6Dh	75h	7Dh
{EBP + disp32}[3]		10	85h	8Dh	95h	9Dh	A5h	ADh	B5h	BDh
ESP/BP/CH		11	C5h	CDh	D5h	DCh	E5h	EDh	F5h	FDh
{ESI}	110	00	06h	0Eh	16h	1Eh	26h	2Eh	36h	3Eh
{ESI + disp8}[3]		01	46h	4Eh	56h	5Eh	66h	6Eh	76h	7Eh
{ESI + disp32}[3]		10	86h	8Eh	96h	9Eh	A6h	AEh	B6h	BEh
ESI/SI/DH		11	C6h	CEh	D6h	DEh	E6h	EEh	F6h	FEh
{EDI}	111	00	07h	0Fh	17h	1Fh	27h	2Fh	37h	3Fh
{EDI + disp8}[3]		01	47h	4Fh	57h	5Fh	67h	6Fh	77h	7Fh
{EDI + disp32}[3]		10	87h	8Fh	97h	9Fh	A7h	AFh	B7h	BFh
EDI/DI/BH		11	C7h	CFh	D7h	DFh	E7h	EFh	F7h	FFh

1 When the Super386 opcode indicates a Gb, Gv, or Gw as one of its operands, the register is specified by the REG field of the MODr/m byte and the operand size, e.g. if the operand size is 16-bit, the Word registers are used.

2 When the Super386 opcode indicates a group instruction, the instruction within the group is specified by the REG (opcode) field of the MODr/m byte, e.g., for Super386 opcode F7h having a MODr/m byte with the REG (opcode) field equal to 100 indicates a MUL.

3 disp8/32 indicates that an 8/32-bit, sign-extended displacement follows the MODr/m byte and must be added to the Base and/or Index Value. For effective addresses using the EBP, the default selector is the SS. All other effective addresses use the DS as the default selector. {— + —} indicates that a SIB byte follows the MODr/m byte.

Table B-10. *Super386 32-Bit Address SIB Encodings*

SIB Byte Format for 32-Bit Addressing Mode			BASE							
			EAX 000	ECX 001	EDX 010	EBX 011	ESP 100	{*}[1] 101	ESI 110	EDI 111
Scaled Index	**Index**	**SCL**	**SIB Values**							
{EAX}	000	00	00h	01h	02h	03h	04h	05h	06h	07h
{EAX*2}		01	40h	41h	42h	43h	44h	45h	46h	47h
{EAX*4}		10	80h	81h	82h	83h	84h	85h	86h	87h
{EAX*8}		11	C0h	C1h	C2h	C3h	C4h	C5h	C6h	C7h
{ECX}	001	00	08h	09h	0Ah	0Bh	0Ch	0Dh	0Eh	0Fh
{ECX*2}		01	48h	49h	4Ah	4Bh	4Ch	4Dh	4Eh	4Fh
{ECX*4}		10	88h	89h	8Ah	8Bh	8Ch	8Dh	8Eh	8Fh
{ECX*8}		11	C8h	C9h	CAh	CBh	CCh	CDh	CEh	CFh
{EDX}	010	00	10h	11h	12h	13h	14h	15h	16h	17h
{EDX*2}		01	50h	51h	52h	53h	54h	55h	56h	57h
{EDX*4}		10	90h	91h	92h	93h	94h	95h	96h	97h
{EDX*8}		11	D0h	D1h	D2h	D3h	D4h	D5h	D6h	D7h
{EBX}	011	00	18h	19h	1Ah	1Bh	1Ch	1Dh	1Eh	1Fh
{EBX*2}		01	58h	59h	5Ah	5Bh	5Ch	5Dh	5Eh	5Fh
{EBX*4}		10	98h	99h	9Ah	9Bh	9Ch	9Dh	9Eh	9Fh
{EBX*8}		11	D8h	D9h	DAh	DBh	DCh	DDh	DEh	DFh
{—}[2]	100	00	20h	21h	22h	23h	24h	25h	26h	27h
{—}[2]		01	60h	61h	62h	63h	64h	65h	66h	67h
{—}[2]		10	A0h	A1h	A2h	A3h	A4h	A5h	A6h	A7h
{—}[2]		11	E0h	E1h	E2h	E3h	E4h	E5h	E6h	E7h
{EBP}	101	00	28h	29h	2Ah	2Bh	2Ch	2Dh	2Eh	2Fh
{EBP*2}		01	68h	69h	6Ah	6Bh	6Ch	6Dh	6Eh	6Fh
{EBP*4}		10	A8h	A9h	AAh	ABh	ACh	ADh	AEh	AFh
{EBP*8}		11	E8h	E9h	EAh	EBh	ECh	EDh	EEh	EFh
{ESI}	110	00	30h	31h	32h	33h	34h	35h	36h	37h
{ESI*2}		01	70h	71h	72h	73h	74h	75h	76h	77h
{ESI*4}		10	B0h	B1h	B2h	B3h	B4h	B5h	B6h	B7h
{ESI*8}		11	F0h	F1h	F2h	F3h	F4h	F5h	F6h	F7h
{EDI}	111	00	38h	39h	3Ah	3Bh	3Ch	3Dh	3Eh	3Fh
{EDI*2}		01	78h	79h	7Ah	7Bh	7Ch	7Dh	7Eh	7Fh
{EDI*4}		10	B8h	B9h	BAh	BBh	BCh	BDh	BEh	BFh
{EDI*8}		11	F8h	F9h	FAh	FBh	FCh	FDh	FEh	FFh

1 If the MOD field of the MODr/m byte equals 00, the BASE is a disp32; otherwise, the BASE is EBP.

2 {—} indicates that the BASE is scaled by the SS amount. This generates the following effective addresses:
{BASE*SCL} (MOD = 00)
{BASE*SCL} + disp8 (MOD = 01)
{BASE*SCL} + disp32 (MOD = 10)

Table B-11. *Super386 Opcode Map*[1]

	0	1	2	3	4	5	6	7
0	ADD Eb, Gb	ADD Ev, Gv	ADD Gb, Ev	ADD Gv, Ev	ADD AL, Ib	ADD eAX, Iv	PUSH ES	POP ES
1	ADC Eb, Gb	ADC Ev, Gv	ADC Gb, Eb	ADC Gv, Ev	ADC AL, Ib	ADC eAX, Iv	PUSH SS	POP SS
2	AND Eb, Gb	AND Ev, Gv	AND Gb, Eb	AND Gv, Ev	AND AL, Ib	AND eAX, Iv	SEG = ES	DAA
3	XOR Eb, Gb	XOR Ev, Gv	XOR Gb, Eb	XOR Gv, Ev	XOR AL, Ib	XOR eAX, Iv	SEG =SS	AAA
4	INC aAX	INC eCX	INC eDX	INC eBX	INC eSP	INC eBP	INC eSI	INC eDI
5	PUSH eAX	PUSH eCX	PUSH eDX	PUSH eBX	PUSH eSP	PUSH eBP	PUSH eSI	PUSH eDI
6	PUSHA	POPA	BOUND Gv, Ev2	ARPL Ew, Gw	SEG =FS	SEG =GS	OP SIZE	ADR SIZE
7	JO Jb	JNO Jb	JB Jb	JNB Jb	JZ Jb	JNZ Jb	JBE Jb	JNBE Jb
8	GRP1 Eb, Ib	GRP1 Ev, Iv	GRP1 Eb, Ib	GRP1 Ev, Ib	TEST Eb, Gb	TEST Ev, Gv	XCHG Eb, Gb	XCHG Ev, Gv
9	NOP	XCHG eCX	XCHG eDX	XCHG eBX	XCHG eSP	XCHG eBP	XCHG eSI	XCHG eDI
A	MOV AL, Ob	MOV eAX, Ov	MOV Ob, AL	MOV Ov, eAX	MOVSB Xb, Yb	MOVSW/D Xv, Yv	CMPSB Xb, Yb	CMPDW/D Xv, Yv
B	MOV AL, Ib	MOV CL, Ib	MOV DL, Ib	MOV BL, Ib	MOV AH, Ib	MOV CH, Ib	MOV DH, Ib	MOV BH, Ib
C	GRP2 Eb,Ib (shift)	GRP2 Ev, Ib (shift)	RET Iw near	RET near	LES Gv, Mp	LDS Gv, Mp	MOV Eb, Ib	MOV Ev, Iv
D	GRP2 Eb,1 (shift)	GRP2 Ev, 1 (shift)	GRP2 Eb, CL (shift)	GRP2 Ev, CL (shift)	AAM	AAD		XLAT
E	LOOPNE Jb	LOOPE Jb	LOOP Jb	JCXZ Jb	IN AL, Ib	IN eAX, Ib	OUT Ib, AL	OUT Ib, eAX
F	LOCK		REPNE	REP REPE	HLT	CMC	GRP3 Eb	GRP3 Ev

1 See legend following Table B-13.

Table B-11. *Super386 Opcode Map (continued)*

8	9	A	B	C	D	E	F	
OR Eb, Gb	OR Ev, Gv	OR Gb, Eb	OR Gv, Ev	OR AL, Ib	OR eAX, Iv	PUSH CS	2nd SET	0
SBB Eb, Gb	SBB Ev, Gv	SBB Gb, Eb	SBB Gv, Ev	SBB AL, Ib	SBB eAX, Iv	PUSH DS	POP DS	1
SUB Eb, Gb	SUB Ev, Gv	SUB Gb, Eb	SUB Gv, Ev	SUB AL, Ib	SUB eAX, Iv	SEG = CS	DAS	2
CMP Eb, Gb	CMP Ev, Gv	CMP Gb, Eb	CMP Ev, Gv	CMP AL, Ib	CMP eAX, Iv	SEG = DS	AAS	3
DEC eAX	DEC eCX	DEC eDX	DEC eBX	DEC eSP	DEC eBP	DEC eSI	DEC eDI	4
POP eAX	POP eCX	POP eDX	POP eBX	POP eSP	POP eBP	POP eSI	POP eDI	5
PUSH Iv	IMUL Gv, Ev, Iv	PUSH Ib	IMUL Gv, Ev, Ib	INSB Yb, DX	INSW/D Yv, DX	OUTSB DX, Xb	OUTSW/D DX, Xv	6
JS Jb	JNS Jb	JP Jb	JNP Jb	JL Jb	JNL Jb	JLE Jb	JNLE Jb	7
MOV Eb, Gb	MOV Ev, Gv	MOV Gb, Eb	MOV Gv, Ev	MOV Ew, Sw	LEA Gv, M	MOV Sw, Ew	POP Ev	8
CBW	CWD	CALL Ap	WAIT	PUSHF	POPF	SAHF	LAHF	9
TEST AL, Ib	TEST eAX, Iv	STOSB Yb, AL	STOSW/D Yv, eAX	LODSB AL, Xb	LODSW/D eAX, Xv	SCASB AL, Xb	SCASW/D eAX, Xv	A
MOV eAX, Iv	MOV eCX, Iv	MOV eDX, Iv	MOV eBX, Iv	MOV eSP, Iv	MOV eBP, Iv	MOV eSI, Iv	MOV eDI, Iv	C
ENTER Dw, Ib	LEAVE	RET Iw far	RET far	INT 3	INT Ib	INTO	IRET	C
ESC	ESC	ESC	ESC	ESC	ESC	ESC	ESC	D
CALL Jv	JMP Jv	JMP Ap	JMP Jb	IN AL, DX	IN eAX, DX	OUT DX, AL	OUT DX, eAX	E
CLC	STC	CLI	STI	CLD	STD	GRP4	GRP5	F

Table B-12. *Super386 Opcode Map With 0Fh Prefix*[1]

	0	1	2	3	4	5	6	7
0	GRP6	GRP7	LAR Gv, Ew	LSL Gv, Ew			CLTS	
1	Active only in SuperState V mode.							
2	MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd	MOV Rd, Td		MOV Td, Rd	
3								
4								
5								
6								
7								
8	JO Jv	JNO Jv	JB Jv	JNB Jv	JZ Jv	JNZ Jv	JBE Jv	JNBE Jv
9	SETO Eb	SETNO Eb	SETB Eb	SETNB Eb	SETZ Eb	SETNZ Eb	SETBE Eb	SETNBE Eb
A	PUSH FS	POP FS		BT Ev, Gv	SHLD Ev, Gv, Ib	SHLDW/D Ev, Gv, CL		
B			LSS Mp	BTR Ev, Gv	LFS Mp	LGS Mp	MOVZX Gv, Eb	MOVZX Gv, Ew
C								
D								
E								
F	Active only in SuperState V mode.							

1 See legend following Table B-13.

Table B-12. *Super386 Opcode Map With 0Fh Prefix (continued)*

8	9	A	B	C	D	E	F	
								0
SCALL Gd, Ed								1
								2
								3
								4
								5
								6
								7
JS Jv	JNS Jv	JP Jv	JNP Jv	JL Jv	JNL Jv	JLE Jv	JNLE Jv	8
SETS Eb	SETNS Eb	SETP Eb	SETNP Eb	SETL Eb	SETNL Eb	SETLE Eb	SETNLE Eb	9
PUSH GS	POP GS		BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL		IMUL Gv, Ev	A
		GRP8	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVESX Gv, Eb	MOVSX Gv, Ew	B
								C
								D
								E
Active only in SuperState V mode.								F

Table B-13. *Super386 Opcode Map for Group Instructions*

GROUP (Opcode)	REG (Opcode) Field of the MODr/m Byte							
	0	1	2	3	4	5	6	7
GRP1[1]	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
GRP2[2]	ROL	ROR	RCL	RCR	SHL	SHR	SHL	SAR
GRP3 (F6/F7)	TEST Eb/Ev, Ib/Iv	TEST Eb/Ev, Ib/Iv	NOT Eb/Ev	NEG Eb/Ev	MUL AH:AL/eDX: eAX, Eb/Ev	IMUL AH:AL/eDX: eAX, Eb/Ev	DIV AH:AL/eDX: eAX, Eb/Ev	IDIV AH:AL/eDX: eAX, Eb/Ev
GRP4 (FE)	INC Eb	DEC Eb						
GRP5 (FF)	INC Ev	DEC Ev	CALL Mp	CALL Ev	JMP Mp	JMP Ev	PUSH Ev	
GRP6 (0F 00)	SLDT Ew	STR Ew	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
GRP7 (0F 01)	SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Ew		LMSW Ew	
GRP8 (0F BA)					BT Ev, Ib	BTS Ev, Ib	BTR Ev, Ib	BTC Ev, Ib

1 Group 1 opcodes are 80h, 81h, 82h, or 83h.

2 Group 2 opcodes are C0h, C1h, D0h, D1h, D2h, or D3h.

The following legend applies to Opcode Tables B-11, B-12, and B-13:

Symbol	Description
Ap	Two operands encoded in instruction. The first one is either 16 or 32 bits, depending on operand size, and the second one is 16 bits.
Cd	32-bit control register.
Dd	32-bit debug register.
Dw	Word sized displacement used by ENTER instruction.
Eb	Byte operand pointed to by the MOD and r/m fields of the MODr/m byte.
Ev	Word or dword operand pointed to by the MOD and r/m fields of the MODr/m byte.
Ev2	Pair of word or dword operands pointed to by the MOD and r/m fields of the MODr/m byte.
Ew	Word operand pointed to by the MOD and r/m fields of the MODr/m byte.
Gb	Byte register pointed to by MODr/m REG field.
Gv	Word or dword register pointed to by MODr/m REG field.
Gw	Word register pointed to by MODr/m REG field.
Ib	Byte immediate encoded in instruction.
Iv	Word or dword operand encoded in instruction.
Iw	Word immediate encoded in instruction
Jb	Byte displacement encoded in instruction relative to instruction address.
Jv	Word or dword displacement encoded in instruction relative to instruction address.
M	Memory address.
Mp	Two operands: the first one is either 16 or 32 bits, depending on operand size, and the second one is 16 bits, pointed to by MODr/m, MOD, and r/m fields.
Ms	Two operands: the first one is 32 bits, and the second one is 16 bits pointed to by the MODr/m, MOD, and r/m fields.
Ob	Byte operand pointed to by displacement encoded in instruction relative to segment base.
Ov	Word or dword operand pointed to by displacement encoded in instruction relative to segment base.
Rd	32-bit register pointed to by the MODr/m REG field.
Sw	Segment selector.
Td	32-bit test register.
Yb	Byte operand pointed to by ES:EDI.
Yv	Word or dword operand pointed to by ES:EDI.
Xb	Byte operand pointed to by DS:ESI.
Xv	Word or dword operand pointed to by ES:ESI.

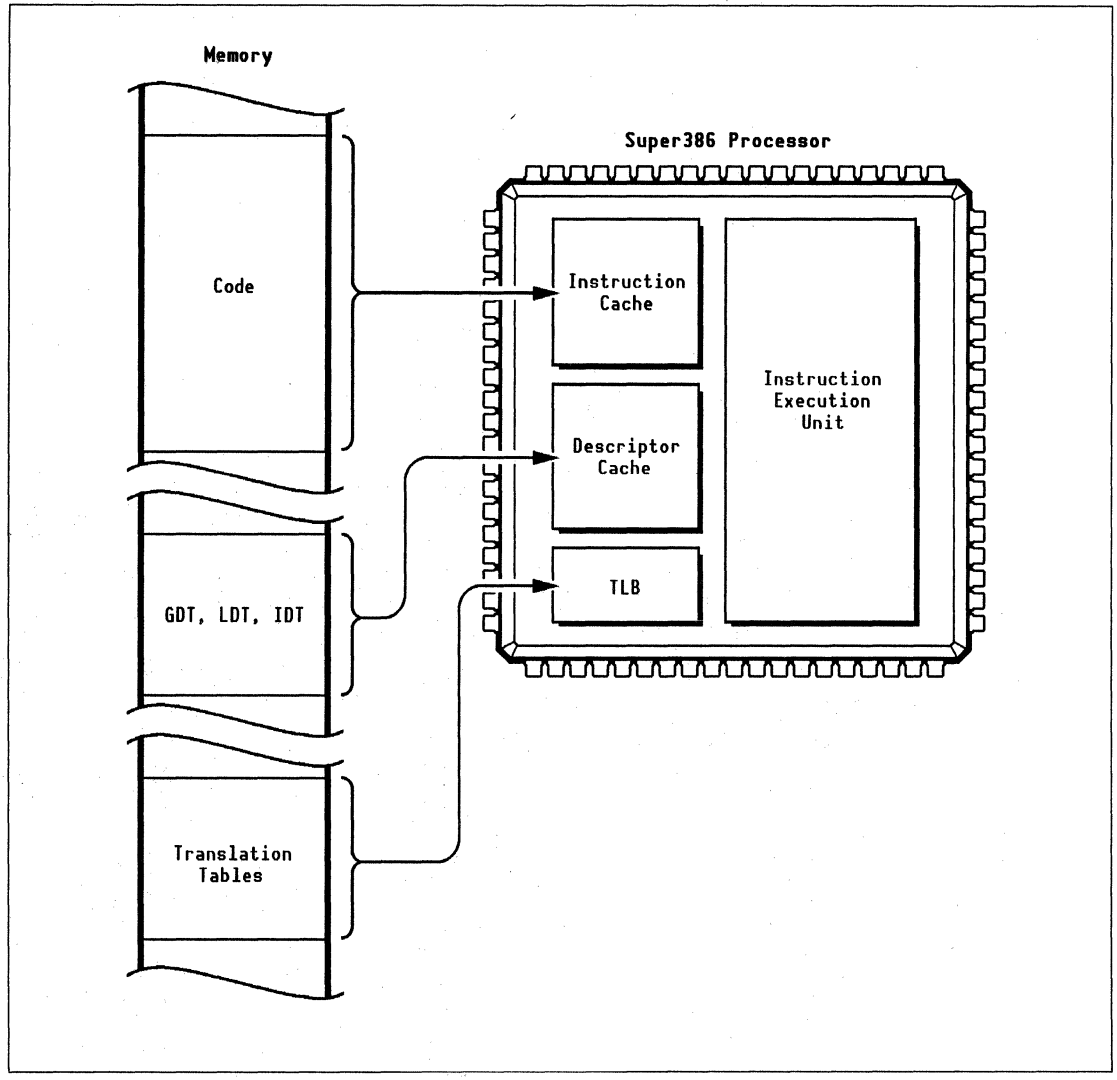
APPENDIX C

Special Programming Considerations

The processor has on-chip registers that enhance performance when you are using frequently referenced data structures in memory. Figure C-1 shows these structures. The 38605DX/DXE processors contain a 512-byte (128-dword) instruction cache that contains previously fetched instructions. The instruction pipeline may have up to four instructions in various stages of processing. When any of the six segment selector registers (CS, DS, SS, ES, FS, and GS) are loaded, their associated shadow register is also loaded automatically by the processor with the descriptor for that segment. The TLB contains up to 32 previously determined linear-to-physical page translations.

These features eliminate the need to refetch instructions, redetermine segment information, and retranslate upon subsequent demand. However, because these on-chip registers hold copies of information stored in memory, several things must be considered before manipulating the memory structures from which the information was copied.

Figure C-1. *On-Chip Data Structure Storage*



The Translation Lookaside Buffer

The TLB is implemented as a four-way set-associative address cache with eight sets, for a total of 32 entries. Upon each generation of a linear address, the TLB is examined to determine if it contains a linear-to-physical address translation entry. Linear address bits 14 to 12 are used to select the TLB set, and the match circuitry determines the appropriate physical-address associate. The TLB updating method ensures that no more than one associate matches a presented linear address. If no associates match, a request is made to the translator to create and place a new entry in the TLB.

Table Filling Mechanism

Because linear address bits 14 to 12 are used to determine the TLB set, and because each set contains four associates, at most four translations can be contained for linear addresses that have identical values for bits 14 to 12. When a fifth translation is required, one of the previous translations must be removed. The hardware determines which translation to remove by using a pseudo-random replacement algorithm. A 2-bit counter is incremented at the end of every memory reference that uses the TLB, and the associate to be replaced is the current value of this counter. The counter is cleared to zero by a hardware reset.

The instruction fetching mechanism can contain at most one translation for the current 4kB page. The execution of the translation occurs independently of the TLB. This translation will satisfy all instruction fetch requests until execution enters another page, or a long displacement potentially causes execution to enter another page. The instruction fetch translation is calculated by the translator and simultaneously written into the TLB. The pseudo-random replacement algorithm may later displace the translation from the TLB without displacing it from the instruction fetching mechanism. At any time, this can allow for as many as five translations to be valid for linear addresses with similar values for bits 14 to 12.

The translator may run even when a translation exists in the TLB. For example, when a memory write is performed, the TLB entry for that area of memory may indicate that the dirty bit in the corresponding page tables in memory is not set; therefore, the translator must be invoked to update this bit in the TLB. The translator can be invoked at other times to revalidate existing translations, or to create translations that the processor predicts will be needed at a future point in time. The translator can also be invoked in some cases when the translation tables are modified.

Because of the manner in which the TLB works, modification of the page translation tables may not necessarily have an immediate effect on the translation process. Also, because of the unpredictability of the pseudo-random replacement algorithm, modification of the translation tables does not always have a predictable delay effect. For example, an attempt to relocate a linear-addressed page by writing the corresponding page table to indicate a different physical address will not have an effect until all translations corresponding to the linear-addressed page are removed from the TLB.

A TLB entry is only written when address translation is enabled and a valid translation is produced from the translation tables. When address translation is disabled, the TLB may or may not retain its previous information. If address translation is again enabled, the previous information may still be valid. The TLB copy of a translation may be retained indefinitely, even though the translation tables have been altered.

Modification of Translation Tables

Linear-to-physical address translation requires two levels of translation—page tables and page directories. Each of the two levels contains a present bit indicating whether the next level is present. For page tables, the present bit indicates whether the page is present in memory. For page directories, the present bit indicates whether the page table (and its associated pages) is present in memory. A page is considered not present if either its page directory or page table indicates that it is not-present. If the page directory indicates that the page is not present, then its associated page table is also considered not present. Creating a page in such a case requires the creation of a page table and the placement of an entry in the page table indicating that the page is present.

By modifying the translation tables in memory, linear-addressed pages can be made present or not present, increased or decreased in protection, or relocated. The time at which the processor begins recognizing these changes depends on the presence of a translation in the TLB, the translation algorithm, and the means by which translations are removed from the TLB.

If a page is made present, the effect will be seen immediately. This is ensured because no entry can as yet exist in the TLB for a nonexistent or not present page. If a decrease in protection is made, that effect will also be seen on the next instruction.

This is also true for normal instruction fetching. When an exception would be reported for a prefetched instruction, the processor is allowed to complete all partially executed instructions before the translation is reattempted. This ensures that all instructions that may have the opportunity to update the translations do so before the translation exception is acknowledged.

If a page is relocated, the translation algorithm will not attempt to revalidate any corresponding translation that the TLB may contain. This will cause some unpredictability as to the time when the relocated translation will take effect. It may take effect immediately, or it may never take effect (if the TLB entry is never displaced). It is also possible that the translation will take effect within an instruction, including the instruction which altered the tables, if that instruction has more than one memory operand.

If a page is made not-present or is increased in protection, the effect also will not be seen as long as an old value corresponding to the linear address remains in the TLB. As with page relocation, the effect can occur immediately. The effect, however, can never occur during the execution of an instruction. To ensure proper operation, any change that could alter a previously established valid translation in the TLB should ensure that such an entry is removed from the TLB before the corresponding page is accessed. All entries in the TLB can be invalidated by reloading the page directory base address in register CR3.

The processor does not prefetch exceptions. The instruction fetch mechanism ensures that all page tables are updated before a translation is attempted. Instruction fetch may or may not query the TLB when attempting to access another page. In cases where it does access the TLB, an indication of an exception causes the translator to revalidate the translation. In cases where it does not access the TLB, the translator will be invoked after all previous instructions have been completed.

Care should be exercised when updating translation tables. Because the TLB may request a translation at almost any time, an intermediate value contained in the tables may cause an otherwise invalid entry to be placed in the TLB and used as if it were valid. This can occur while the instructions doing the updates are still executing, or at any time when the tables are in an inconsistent state.

Insertion of an invalid entry in the TLB is also a concern during enabling of paging. The PG bit of CR0 is examined at each generation of a linear address to determine whether translation should take place. Setting the PG bit to 1 may cause the translation to begin on the fetch of the very next instruction, or it may delay the translation until some unpredictable number of instructions have begun execution.

Translation will begin for the next operand fetch, which ensures that all following operands are accessed with translation on. But the presence of the instruction prefetch queue and the instruction cache allows some instructions, that were fetched before paging was enabled, to be available for execution. To ensure that translation takes effect, a jump instruction to the new linear-addressed page where execution continues should be executed immediately after paging is enabled in order to empty the prefetch queue. Because translation may become active on the fetch of the required jump instruction, the translation tables should be initialized to contain an entry for the page containing the paging enable code. This entry should indicate an identity mapping in which the linear and physical addresses are identical.

Page Aliasing

There are no restrictions on page aliasing. Translation tables can be constructed to cause multiple linear addresses to map to a single physical page. When this is done, multiple translation paths read to a single physical page, complicating the use of the reference and dirty information in the tables. Because this information is somewhat linear-address dependent, it is necessary to examine all the translation entries for each linear address range to determine whether a physical page has been altered or referenced. It is also possible to support inconsistent levels of protection. Two linear address ranges can map to the same physical address. One range may provide a different kind of protection than another. An operating system that determines which pages to deallocate must be aware of all the aliases by which each physical page can be accessed.

Validating Multiple Translations

The execution of some instructions requires the validation of more than one translation. This occurs most often for operands that cross page boundaries. Such operands have an upper and a lower linear addressed page. The processor detects such operands and validates the translations for the two pages before generating bus accesses. The order in which these validations occur may not be the same as the order in which the portions of the operands are accessed.

For example, the lower page may be checked for translations before the upper page, but the upper page may be accessed first. Accesses generated on the bus by the translator may be for the page that will not be accessed until some undefined number of unrelated bus cycles have occurred. External hardware should not depend on translation accesses to indicate which operands will be accessed in the immediate future.

Another example is the INS instruction, which reads a value from an I/O port and writes it to a memory location. The memory location is examined to ensure that it has a valid translation before the request is made to the I/O port. This ensures that the value returned from the port can be placed in the destination immediately. Because I/O devices function differently from memory, and because it is valid for an I/O device to return different data for each read from the same port number, any attempt to re-execute an instruction that has already retrieved a value from the port may result in the loss of the value first retrieved. This is not a problem with memory locations because they return the same value as long as the location remains unmodified.

Exceptions

If an exception is encountered in the translation process, either for a portion of a page-crossing operand or for the destination of the INS instruction, the software page-fault handler will be invoked. In all cases, the instruction causing the page fault must do so before it has modified the state of any memory, I/O device, or CPU register. This allows the instruction to be re-executed. The address reported in register CR2 may not correspond to the first address accessed by the instruction, which may encounter a page fault. In the case of the page-crossing operand where both portions encounter a page fault, the first address reported will be the last address actually accessed.

Addresses Not Translated

Some addresses are not subject to translation. These include addresses generated for I/O accesses and accesses to the translation tables. Translation is never active in real mode. Future implementations of the architecture may increase the number of entries in the TLB or may use a different replacement algorithm.

Segment Descriptors

The virtual-memory environment created by segmentation is much coarser than that created by paging. Software can use up to six segments at any one time, but may use thousands of pages. Segmentation faults are encountered only by instruction fetches or operand accesses that either exceed the segment limit or violate segmentation rules. Page faults, on the other hand, can be encountered for any page of any segment. Segment descriptors contain the information needed for translating effective addresses to linear addresses. They are loaded automatically by the processor whenever software loads segment selectors into the segment selector registers. With the segment descriptors loaded, no additional segmentation information is needed for processing.

Each segment descriptor contains a base, limit, and protection information for the segment. A segment selector register is loaded only when a MOV segment, POP segment, interrupt, exception, far JMP, far CALL, or protected-mode gate is encountered. No replacement algorithm is used for the descriptors that are automatically loaded. Old entries are simply replaced by new entries. A return to the original segment requires a reload of the appropriate segment selector register.

In Real and Protected Modes

In real mode, a segment descriptor is equal to the segment selector shifted left by four bits, and no tables are used to obtain the descriptor. In protected mode, a segment descriptor is contained in the GDT or LDT; the selector for the segment indicates the appropriate table and the entry within the table. The segment descriptors in memory contain an accessed bit (bit 8) which is set to 1 by the processor when the descriptor is loaded into its shadow register. To do this, the processor performs a locked update on the descriptor's appropriate doubleword in the selected descriptor table. The processor may perform this operation even if the accessed bit is already set.

Because no table is used in real mode, there are no consistency considerations between it and the contents of the segment shadow registers. In protected mode, however, the segment shadow registers represent a subset of the information in the descriptor tables, and modification to the tables in memory requires awareness of consistency considerations similar to those affecting the TLB.

Descriptor Table Modification

A modification to the descriptor tables is not reflected in a segment shadow register until the selector pointing to the modified entry is reloaded by software. Attempts to increase the size of the data segment, for example, by increasing the descriptor limit in the table do not have an immediate effect. The segment shadow register will continue to contain the old limit value, and exceptions will be generated for any operand exceeding it. If the limit is increased transparently to the executing program, the code that increases the limit must also reload the segment selector register.

The point in time when changes to descriptors are reflected in the processor is predictable: old segment descriptors are retained by the processor until they are explicitly loaded by software. The contents of the TLB, by comparison, is not so predictable: translations may be displaced by the LRU replacement algorithm at any time. The delay associated with loading a descriptor shadow register is accounted for in the clock counts of the instruction that caused the loading. A page translation, on the other hand, may be required for each memory operand of any instruction, accounting for the range of clock counts quoted for instructions. The execution time of instructions is therefore more predictable when paging is disabled.

Segment Aliasing

As with page translation, it is possible to support aliases using segmentation. This may be useful in protected mode, for example, when an executing program needs to modify data in the code segment. Protected mode normally prohibits modifications of the code segment, but by aliasing the code and data segments, a write to the data segment will update the identical location in the code segment.

It is possible to implement partially overlapping aliases as well. The stack segment, for example, may begin in the middle of the data segment and extend to the end of the data segment. This makes the stack segment a subset of the data segment, while still allowing the data segment direct access to the stack. If the operating system deallocates segments, it must be aware of all the users of each shared segment.

The 38605DX/DXE Instruction Cache

The 38605DX/DXE processors have a 512-byte instruction cache to increase processor performance. The cache contains 128 directly mapped doublewords, each of which has tag information allowing it to map to any address value for bits 31 to 9. Because the cache is directly mapped, no special replacement algorithm is used. Old entries are simply replaced by new entries.

On average, about 65 percent of all instruction fetches are satisfied by the cache. The actual cache hit rate varies dramatically, from zero to 100 percent, depending on the nature of the executing code. When instruction data is available from the cache, the external bus is available for operand accesses. Four bytes can be read from the cache in a single cycle, or eight bytes in the equivalent of one bus access. Special hardware is also included to generate addresses for jump instructions. In some cases, this combination of cache and hardware address generation dramatically increases execution speed. Programs that are unusually sensitive to execution speed may execute too fast when the cache is enabled. In these cases, the cache may need to be disabled.

Cache Consistency Mechanism

A cache consistency mechanism is required to ensure that instructions contained in the cache, as well as those currently executing in the processor pipeline, accurately reflect the contents of memory. To this end, the 38605DX/DXE and the 38600DX/DXE processors contain identical consistency checking hardware. This hardware functions on the 38605DX/DXE processors whether the cache is

enabled or not. Because instructions may be present in the pipeline without being present in the cache, the 38600DX/DXE processors also implement the same consistency mechanism.

The consistency mechanism functions by keeping a record of instructions contained in the cache or pipeline. When a store is executed to an address that matches one of those recorded by the mechanism, a pipeline serialization operation is generated. This flushes the prefetch queue, and the subsequent code fetches retrieve the updated information. Because of pipeline latency, it is not possible to prevent the instructions immediately following the store from completing. Only the second (and subsequent) instructions following the store instruction will be refetched. This reduces the unobservable nature of code-space stores to a small amount.

Programs that attempt to determine the size of the prefetch queue by storing ahead in the instruction stream will indicate a small to nonexistent queue. The nondecoded prefetch queue is 12 bytes long. The consistency mechanism causes it to appear smaller, but the performance benefits of a 12-byte queue are fully realized.

The consistency mechanism keeps track of physical addresses. This ensures that stores to code space by way of segment or translation synonyms function exactly as if no synonyms were used. External devices that store to memory located in the instruction cache cause the corresponding cache entry to be invalidated and the instruction queue to be flushed.

Future implementations of the architecture may take greater liberties as to when modifications to the code segment are reflected in the code stream. A cache consistency mechanism will be present in all implementations, but the nature of future pipelines may increase the number of instructions that must execute before the change is observable. In general, a programmer wanting to effect a change in the code segment should execute a jump instruction to flush the prefetch queue.

Enabling and Disabling the Instruction Cache

The 38605DX/DXE processors can be operated with the instruction cache enabled or disabled. When the instruction cache is enabled, each code fetch is written into the cache and the entry is made valid. Future accesses to the same address will retrieve the data from the cache. When the cache is disabled, its contents may or may not be displaced over time. Software cannot depend on the contents of the cache being retained while it is disabled. Similarly, software cannot assume that the entire content of the cache is invalidated by the act of disabling it. To invalidate the entire cache, the FLUSH* pin must be asserted. Invalidating the cache from software is unnecessary because the consistency mechanism ensures that the cache always reflects an exact copy of what is in memory.

The Instruction Fetching Mechanism

Instructions can be one to 15 bytes in length. The act of fetching the instruction does not correspond to the size of the instruction. A 1-byte INC instruction may not generate a 1-byte fetch. Similarly, a 15-byte instruction will not generate a single fetch of a 15-byte quantity. Each processor supports a maximum fetch size, and nearly all instruction fetches return this maximum amount of data. The 38600DX/DXE processors support a 4-byte fetch every two cycles. The 38605DX/DXE processors support a 4-byte fetch every cycle from the instruction cache. Future implementations may support greater maximums.

Because instructions are prefetched, not all that are fetched will be executed. Many prefetched instructions are discarded when taken jumps are encountered. Future implementations may support the concept of speculative execution, allowing instructions following jumps to be prefetched before it is known whether the jumps are taken or not. This will increase the number of discarded prefetches and cause their addresses to be randomly distributed.

Instructions are prefetched only when no segmentation or page-translation violation would be encountered. If such a violation were to occur in a prefetch, the instruction fetching mechanism would wait until the processor had a chance to complete all partially executed instructions before attempting the fetch. This would allow any previous instruction to resolve the violation.

Instructions may be fetched in an order that is different from the order in which they appear to execute. This happens most often in the 38605DX/DXE processors when the cache is enabled. An example is a code fetch following a loop. Prefetch may decide to fetch the instruction data upon first encountering the loop. This prefetch will return instruction data that is not required immediately because the loop was taken but may still be written into the instruction cache. When it is in the cache, no further code is fetched from the same address. The observed effect is that the code fetch for the instruction occurs on the first iteration of the loop instead of the last, as would normally be expected.

In some cases, an instruction can be fetched many times. This may occur, for example, when an exception is predicted to revalidate the exception status. When instructions are fetched many times, any alteration of the location by another device may or may not cause the instruction to be interpreted as modified.

The instruction prefetch queue can contain up to 12 bytes of instruction data. This could be as many as 12 instructions or less than one. The prefetch mechanism may generate a fetch for the bytes that would normally be placed into the queue, before any room is available there. This is done under the assumption that by the time the fetch returns data, room will have become available. If the queue is still full in such a case, the fetched data is discarded. The combination of the queue and the ability

to fetch beyond the end of the queue have the effect on the processor of making the prefetch queue appear to external devices to be as much as 16 bytes in size. Future implementations may increase this size without limit.

Sequence of Storage References

From an assembly language programmer's viewpoint, the processor executes instructions sequentially. The execution of one instruction precedes the execution of the next. Within each instruction, operands appear to be accessed in a defined order. The PUSH memory instruction, for example, first fetches the operand at the memory location, then places it on the stack by storing it to a location indicated by the stack pointer. The descriptions of instructions in Appendix A, "Instruction Set Reference," indicate the order in which operands appear to be accessed.

Factors Influencing the Order of Instruction Fetches

The processor may, at times, execute instructions and access operands in an order other than already described. This may be done, for example, to ensure that no exceptions are present on particular operands or to speed the execution of an instruction. In other cases, an instruction that conceptually follows another instruction may complete execution before the first instruction finishes. In all cases, the appearance of the processor's operation is guaranteed to agree with the conceptual sequence. However, devices external to the processor may observe a sequence of operations different from the conceptual sequence.

Unaligned operands require multiple bus accesses to fetch or store them. This can present a problem if an external device is allowed to observe an unaligned operand between the time when the first portion is modified and the the second portion is modified. Also, because instructions with multiple memory operands may store them in a nonconceptual order, the consistency of such operands is unreliable. The PUSHA instruction, for example, may not write the entries to the stack in the conceptual order, starting with EAX and ending with EDI. An external device may be in error if it assumes that new values have been written for all registers when a new value is observed for EDI.

Instruction Execution Ordering

The processor allows the execution of instructions to continue while the bus is busy processing a request. A write to a slow memory device, for example, may keep the bus busy for an extended period of time. When this happens, the processor is allowed to execute instructions that follow, providing that these conditions are met:

- The memory access is to a present page, and no page protection faults are detected.
- The memory access does not violate any segmentation protection rules.
- The memory access is the last operation performed by the instruction.
- The following instructions do not request a pipeline serialization operation.
- The following instructions do not depend on the data returned by the memory access.
- The following instructions do not themselves require the bus.

No limit is placed on the number of instructions that follow the instruction using the bus. Execution can continue for as long as the above rules are observed. An instruction that contains one or more memory operands, but does so only late in its execution, will execute up to the point where it requires the bus. The 38600DX/DXE processors limit this overlap to the number of instructions that can be present in the pipeline following the instruction generating the slow memory access (3), plus the number of instructions present in the prefetch queue (12). The 38605DX/DXE processors impose no absolute limit because the instruction cache allows for the execution of loops. Jump instructions do not require use of the bus when their target code is available from the instruction cache.

Instruction-Fetch Reordering

The 38605DX/DXE instruction cache alters the conceptual order of operand accessing to the extent that it reduces the number of code prefetches. It is common to execute code that generates no code prefetches for many thousands of cycles. This is not true when the cache is disabled, and does not apply to the 38600DX/DXE processors.

The ability to continue execution while a preceding instruction is still performing a store or fetch does not alter the conceptual order of operand access, but it can have a dramatic effect on the time between operand accesses. It is possible for a minimum delay between accesses of 100 cycles to be reduced to zero cycles by the act of enabling the cache. This might occur, for example, if a store to a slow memory device is followed by a series of instructions within a loop, none of which access memory. While the store operation is waiting, the following instructions execute

until one of them requires the bus. When such an instruction is encountered, it will wait until the store completes, and it may appear on the bus during the cycle immediately following the store.

The 38605DX/DXE processors' ability to overlap instruction execution in this manner differs from the similar ability of a write buffer, because the 38605DX/DXE can also overlap slow fetches. Instructions are allowed to continue execution following a slow fetch as long as the operand being fetched is not required by the instructions immediately following. Special register consistency hardware contained in the processor ensures that when an instruction requiring this operand enters the pipeline, it is held until the fetch is complete.

Certain operations require the pipeline to be serialized. I/O accesses ensure that the actual sequence matches the conceptual sequence by forcing the pipeline to complete all other memory operations first, and then delaying all further operations until the I/O access is complete. Instructions that alter the ability to accept interrupts—either enabling or disabling them (STI, CLI, POPF, task switches, and IRET)—also perform a serialization.

Future implementations of the architecture may hide even greater deviations from the conceptual sequence of operand accesses. Instructions or operands may be fetched in nonsequential order, or may be fetched in smaller or larger pieces than the conceptual picture indicates. The architecture will ensure that the conceptual sequence appears to be followed from the processor's viewpoint, but external devices may be exposed to the changes. To allow for full compatibility with future processors, any external device that is sensitive to the order in which operands are accessed must use semaphores.

The result of some instructions depends on the order in which operands are accessed. This is true for the REP MOVS instruction. A source operand that overlaps the destination operand will alter the data being moved. It is possible for the overlap to occur so that the source is retrieved from the previous iteration's destination. In such a case, a future implementation may allow the processor to determine that the destination is identical for all iterations and alter the algorithm to eliminate the fetches normally required. The single destination value is simply stored upon each iteration. Again, this does not alter the conceptual sequence of operations, but to an external device, memory reads disappear that would otherwise be observed.

Similarly, a future implementation may allow each repeated string iteration to be consolidated into fewer iterations of larger quantities of data. A REP SCASB instruction may be reduced to a quarter the number of iterations by interpreting it as a special version of the REP SCASD instruction. In such an implementation, it is unpredictable whether the operands are accessed as a byte, word, or doubleword at a time. The order in which operands are accessed may also be unpredictable. No matter how the actual sequence is altered, exception recognition will appear to be identical to the conceptual picture.

The processors' ability to hold one memory access pending, while the following instructions continue execution, may delay stores for an undefined period of time. Future implementations may include store buffers that can hold more than one store pending. There is no limit on the length of time that fetches or stores may be held pending. It is possible for a pending store to be passed to a following fetch without updating external memory by passing the value internally in the processor.

Operands are accessed in order, according to the rules listed in Table C-1.

Table C-1. *Operand Accessing Rules*

Always in Order	May or May Not Be in Order	Usually Out of Order
Stores between multiple instructions	Fetches between multiple instructions	Instruction fetches
Fetches and stores that precede or follow an I/O-space access	Fetches and stores within instructions	
Fetches and stores that precede or follow an interrupt enabling or disabling operation		

Semaphore Locking

Systems with more than one bus master must allow the use of semaphores. Semaphores are used to communicate information between bus masters. To function properly, they must support a read-modify-write operation as an indivisible sequence. If this were not the case, another bus master could perform the modification portion of the operation after the first bus master read the value. In addition, as mentioned above, compatibility with future processors may only be possible if semaphores are used with external devices that are sensitive to the order in which operands are accessed.

The processor supports semaphores by providing the LOCK* signal. This signal is asserted explicitly when the LOCK instruction prefix is executed. It is asserted implicitly by page-table or descriptor updates, or by the XCHG instruction. When asserted, external system hardware architecture prohibits bus masters other than the one requesting the LOCK* from accessing the bus. For the typical read-modify-write operation, LOCK* will be asserted when the operation begins, and will remain asserted until the modification completes.

Within certain bounds, the external system may lock specific memory regions corresponding to the address presented on the bus when the LOCK* is first asserted. An unaligned bus access may generate an additional access outside the range of the first access to satisfy the alignment requirements. A simple system would lock the entire memory region, but this may result in troublesome performance consequences.

Any system that attempts to lock specific memory locations, however, must also lock adjacent doubleword-aligned doublewords. When the first access is generated, it is unpredictable whether a second access will be required. Because page translation can alter the appearance of "adjacent" locations, any system that supports demand paging must also require the locking mechanism to ignore bits 31 to 12 of the address.

Glossary

A

AADS*	Alternate address space signal (output). Initiates a bus cycle in the SuperState V mode (38605DXE only).
AF	Auxiliary carry flag (bit 4 of EFLAGS register)
ALU	Arithmetic logic unit
ANMI*	Alternate non-maskable interrupt signal; indicates a request to process a SuperState V interrupt (input pin, 38605DXE processor only).

B

BP	Base pointer (GPR)

C

CF	Carry flag (bit 0 of EFLAGS register)
CPL	Current privilege level, determined by the processor and stored in the RPL field of the CS selector register.
CR*n*	Control register CR3, CR2, or CR0
CS	Code segment register

D

D31:0	Designates the 32 lines in the data bus.
DF	Direction flag (bit 10 of EFLAGS)
DI	Destination index (GPR)
DPL	Descriptor privilege level, stored in a segment's descriptor
DS	Base-address register of an active data segment

E

EAR	Effective-address register
EAX	Accumulator register (GPR)
EBP	Base pointer register (GPR)
EBX	Base register (GPR)
ECX	Count register (GPR)
EDI	Destination index register (GPR)
EDX	Data register (GPR)

EFLAGS	32-bit flag register
EIP	Instruction-pointer register
ERROR*	Coprocessor error signal generated in previous instruction and nc masked.
ESI	Source index register (GPR)
ESP	Stack pointer register (GPR)

F

FLAGS	Lower 16 bits of EFLAGS register
FLUSH*	Cache flush signal (input pin, 38605DX/DXE processors only)

G

GDT	Global descriptor table
GDTR	Global descriptor table register
GPR	A general-purpose register. Acronyms that are marked (GPR) in this glossary represent general-purpose registers.

I

IDT	Interrupt descriptor table
IDTR	Interrupt descriptor table register
IF	Interrupt (INTR) enable flag (bit 9 of EFLAGS)
INTR	Maskable interrupt request signal (input)
IOPB	I/O privilege bitmap, located in the TSS
IOPL	I/O privilege level (bits 12 and 13 of EFLAGS)

K

KEN*	Cache enable signal (input pin, 38605DX/DXE processors only)

L

LDT	Local descriptor table
LDTR	Local descriptor table register
LOCK	An instruction prefix that guarantees that the processor retains control of the bus during the execution of the instruction. It asserts the LOCK* signal.

M

MSW	Machine status word, the lower word of the CR0 register

N

NMI	Nonmaskable interrupt
NT	Nested task bit (bit 14 of the EFLAGS register)

O

OF	Overflow flag bit (bit 11 of the EFLAGS register)

P

PDBR	Page directory base register in the CR3
PF	Parity flag; bit 2 of EFLAGS register

R

RESET	Microprocessor general reset
RF	Resume flag bit (bit 16 of the EFLAGS register)
RPL	Requestor privilege level, stored in a segment's selector

S

SF	Sign flag bit (bit 7 of the EFLAGS register)
SI	Source index (GPR)
SIB	Scale, index base byte
SP	Stack pointer (GPR)
SS	Current stack segment register

T

Task	A protected-mode environment in which programs (and their procedures) can run at one of four privilege levels. Each task has its own segment in memory, called a task state segment (TSS), which is accessed via a descriptor contained in the GDT.
TF	Trap enable flag (bit 8 of the EFLAGS register)
TI	Table indicator bit in selector field; 1 selects local descriptor table; 0 selects global descriptor table.
TLB	Translation lookaside buffer
TR	Task register
TSS	Task state segment. It contains a copy of all the registers and values that must be saved to preserve the state of a task when switching between tasks. The contents of the CS and EIP registers are saved separately for privilege levels 0, 1, and 2.

V

VM	Virtual-8086 mode bit (bit 17 of EFLAGS register)
VM86	Virtual-8086 mode

Z

ZF	Zero flag (bit 6 of the EFLAGS register)

Index

B

C

D

E

F

M

N

O

P

S

T

U

V

W

X

Z

ional Sales Offices

United States

California, Irvine
Chips and Technologies, Inc.
Phone: 714-852-8721

California, San Jose
Chips and Technologies, Inc.
Phone: 408-437-3300

Georgia, Norcross
Chips and Technogies, Inc.
Phone: 404-662-5098

Illinois, Schaumburg
Chips and Technologies, Inc.
Phone: 708-397-4300

Massachusetts, Andover
Chips and Technologies, Inc.
Phone: 508-688-4600

Texas, Dallas
Chips and Technologies, Inc.
Phone: 214-702-9855

International

Germany, Munich
Chips and Technologies, GmbH
Phone: 011-49-89-46-3074

Hong Kong
Chips and Technologies, Inc.
Phone: 011-852-5-980010

Japan, Tokyo
Chips and Technologies, Japan K.K.
Phone: 011-81-3-379-74311

Korea, Seoul
Chips and Technologies, Inc.
Phone: 011-82-2-558-5559

Switzerland, Marin
Chip and Technologies, S.A.
Phone: 011-41-3-8336379

Taiwan, Taipei
Chips and Technologies, Inc.
Phone: 011-886-2-717-5595

United Kingdom, Berkshire
Chips and Technologies, UK
Phone: 011-44-734-880237

Sales Representatives

United States

Alabama, Huntsville
B.I.T.S.
Phone: 205-859-2686

Alabama, Huntsville
Reptron Electronics, Inc.
Phone: 205-722-9500

Arizona, Scottsdale
AzTECH Component Sales
Phone: 602-991-6300

California, Orangeville
Magna Sales
Phone: 916-989-0843

California, Santa Clara
Magna Sales
Phone: 408-727-8753

California, San Diego
S.C. Cubed
Phone: 619-458-5808

California, Thousand Oaks
S.C. Cubed
Phone: 805-496-7307

California, Tustin
S.C. Cubed
Phone: 714-731-9206

Colorado, Wheat Ridge
Wescom Marketing, Inc.
Phone: 303-422-8957

Connecticut, Guilford
DataMark Inc.
Phone: 203-453-0575

Florida, Casselberry
Dyne-A-Mark Corp.
Phone: 407-831-2811

Florida, Clearwater
Dyne-A-Mark Corp.
Phone: 813-441-4702

Florida, Ft. Lauderdale
Dyne-A-Mark Corp.
Phone: 305-771-6501

Georgia, Norcross
B.I.T.S., Inc.
Phone: 404-446-1155

Idaho, Boise
Wescom Marketing, Inc.
Phone: 208-336-6654

Illinois, Hoffman Estates
Micro-Tex, Inc.
Phone: 312-765-3000

Indiana, Carmel
Giesting & Associates
Phone: 317-844-5222

Kentucky, Versailles
Giesting & Associates
Phone: 606-873-2330

Maryland, Annapolis
EES
Phone: 410-269-4234

Massachusetts, Woburn
Mill-Bern Assoc.
Phone: 617-932-3311

Michigan, Coloma
Giesting & Associates
Phone: 616-468-3308

Michigan, Comstock Park
Giesting & Associates
Phone: 616-784-9437

Michigan, Livonia
Giesting & Associates
Phone: 313-478-8106

Minnesota, Eden Prairie
High Tech Sales Assoc.
Phone: 612-944-7274

Missouri, Bridgeton
Centech
Phone: 314-291-4230

Missouri, Raytown
Centech
Phone: 816-358-8100

New Jersey, Morristown
T.A.I.
Phone: 609-778-5353

New York, Commack
ERA, Inc.
Phone: 516-543-0510

New York, Pleasant Valley
Pitronics
Phone: 914-635-3233

New York, Syracuse
Pitronics
Phone: 315-455-7346

New York, Williamsville
Pitronics
Phone: 716-689-2378

North Carolina, Raleigh
B.I.T.S., Inc.
Phone: 919-676-1880

Ohio, Cincinnati
Giesting & Assciates
Phone: 513-385-1105

Ohio, Cleveland
Giesting & Associates
Phone: 216-261-9705

Oregon, Beaverton
L-Squared Limited
Phone: 503-629-8555

Pennsylvania, Pittsburg
Giesting & Associates
Phone: 412-828-3553

Texas, Austin
OM Associates, Inc.
Phone: 512-794-9971

Texas, Houston
OM Assoc., Inc.
Phone: 713-789-4426

Texas, Richardson
OM Assoc., Inc.
Phone: 214-690-6746

Utah, Salt Lake City
Wescom Marketing, Inc.
Phone: 801-269-0419

Washington, Kirkland
L-Squared Limited
Phone: 206-827-8555

Wisconsin, Waukesha
Micro-Tex, Inc.
Phone: 414-542-5352

Canada

Ontario, Kanata
Electro Source, Inc.
Phone: 613-592-3214

Ontario, Rexdale
Electro Source, Inc.
Phone: 416-675-4490

Quebec, Pointe Claire
Electro Source, Inc.
Phone: 514-630-7846

British Columbia, Vancouver
Electro-Source, Inc.
Phone: 604-435-8066

ributors

n America

ted States

zona, Tempe
hem
ne: 602-966-6600

lifornia, Chatsworth
nthem
none: 818-755-1333

California, Fountain Valley
Bell Microproducts
Phone: 714-963-0667

California, Irvine
Anthem
Phone: 714-768-4444

California, Milpitas
Bell Microproducts
Phone: 408-434-1150

California, Rocklin
Anthem
Phone: 916-624-9744

California, San Diego
Anthem
Phone: 619-453-9005

California, San Jose
Anthem
Phone: 408-453-1200

Colorado, Englewood
Anthem
Phone: 303-790-4500

Connecticut, Waterbury
Anthem
Phone: 203-237-2282

Florida, Boca Raton
JACO
Phone: 407-241-7943

Florida, Ft. Lauderdale
Reptron Electronics, Inc.
Phone: 305-735-1112

Florida, Tampa
Reptron Electronics, Inc.
813-854-2351

Georgia, Norcross
Reptron Electronics, Inc.
Phone: 404-446-1300

Georgia, Norcross
JACO
Phone: 404-449-0275

Illinois, Elk Grove
Anthem
Phone: 708-884-0200

Illinois, Shaumburg
Reptron Electronics, Inc.
Phone: 312-882-1700

Massachusetts, Wilmington
Anthem
Phone: 508-657-5170

Massachusetts, Wilmington
Bell Microproducts
Phone: 508-658-0222

Maryland, Columbia
Anthem
Phone: 301-995-6640

Maryland, Columbia
JACO
Phone: 301-995-6620

Michigan, Livonia
Reptron Electronics, Inc.
Phone: 313-525-2700

Minnesota, Eden Prairie
Anthem
Phone: 612-944-5454

Minnesota, Minnetonka
Reptron Electronics, Inc.
Phone: 612-938-0000

New Jersey, Pinebrook
Anthem
Phone: 201-227-7960

New York, Commack
Anthem
Phone: 516-864-6600

New York, Hauppauge
JACO
Phone: 516-273-5500

North Carolina, Raleigh
Reptron Electronics, Inc.
Phone: 919-870-5189

JACO
Phone: 919-876-7767

Ohio, Columbus
EMC
Phone: 614-299-4161

Ohio, Solon
Reptron Electronics, Inc.
Phone: 216-349-1415

Ohio, Worthington
Reptron Electronics, Inc.
Phone: 614-436-6675

Oklahoma, Tulsa
JACO
Phone: 918-664-8812

Oregon, Beaverton
Anthem
Phone: 503-643-1114

Pennsylvania, Horsham
Anthem
Phone: 215-443-5150

Texas, Addison
JACO
Phone: 214-733-4300

Texas, Austin
JACO
Phone: 512-835-0220

Texas, Richardson
Anthem
Phone: 214-238-7100

Texas, Richardson
All-American
Phone: 214-231-5300

Texas, Sugarland
All-American
713-530-0958

Texas, Sugarland
JACO
Phone: 713-240-2255

Utah, Salt Lake City
Anthem
Phone: 801-973-8555

Washington, Bothell
Anthem
Phone: 206-483-1700

Canada

Ontario, Woodbridge
Valtrie Marketing
Phone: 416-798-2555

Distributors

International

Asia/Pacific

Australia, Sydney
ZATEK Components
Phone: 011-61-2-8740122

Hong Kong, Kwung Tong
Wong's KK Ltd.
Phone: 011-852-3450121

India, Bombay
Silicon Electronics
Phone: 011-91-22-243460

India, New Dehli
Ajay Jain
Phone: 011-91-11-6863044

Israel, Tel-Aviv
CVS
Phone: 011-972-3-5447475

Japan, Kawasaki
CTC Components Systems Co., Ltd.
Phone: 011-81-44-8525121

Japan, Tokyo
ASCII and Mitsui and Company
Phone: 011-81-33-5022251

Korea, Seoul
Nae Wae Semiconductor
Phone: 011-82-2-8429500

Malaysia, Penang
Dynamar
Phone: 011-60-4-363376

Singapore
Technology Distribution
PTE Ltd.
Phone: 011-65-2997811

Taiwan, Taipei
Ally, Inc.
Phone: 011-886-2-7886270

Taiwan, Taipei
World Peace
Industrial Co., Ltd.
Phone: 011-886-2-7865311

Thailand, Bangkok
Grawinner
Phone: 011-66-2-2158742

Europe

Belgium, Zaventem
ACAL Auriema Belguim
Phone: 011-32-2-7205983

Denmark, Herlev
Nordisk, Elektronik A/S
Phone: 011-45-4-2842000

Finland, Helsinki
OY Fintonic AB
Phone: 011-358-0-6926022

France, Le Chesnay
A2M
Phone: 011-33-1-39549113

Germany, Nettetal
Rein Elektronik GmbH
Phone: 011-49-2153-7330

Italy, Milano
Moxel S.R.L
Phone: 011-39-2-61290521

Netherlands, Eindhoven
ACAL Auriema Nederland B.V.
Phone: 011-31-40-816565

Norway, Hvalstad
Nordisk Elektronik A/S
Phone: 011-47-2846210

Spain, Madrid
Compania Electronica de Tecnicas
Aplicadas, S.A.
Phone: 011-34-1-7543001

Spain, Barcelona
Compania Electronica de Tecnicas
Aplicadas, S.A.
Phone: 011-34-3-3007712

Sweden, Kista
Nordisk Elektronik A.B.
Phone: 011-46-8-7034630

Switzerland, Dietikon
DataComp AG
Phone: 011-41-1-7405140

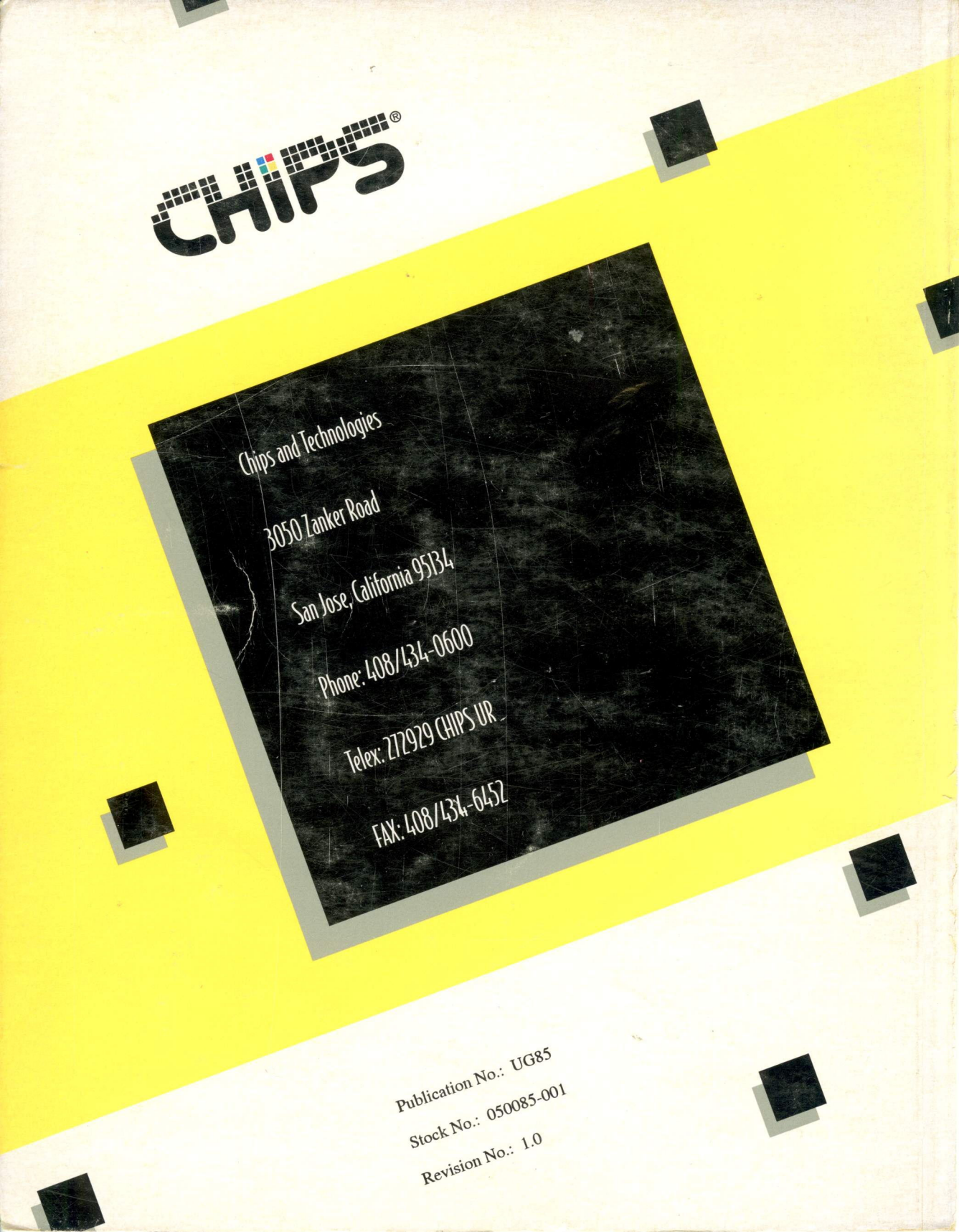
United Kingdom, Berkshire
Magna Technology
Phone: 011-44-734-880211

Sirretta Microelectronics, Ltd.
Phone: 011-44-734-311822

United Kingdom, Oxfordshire
Thame Components, Ltd.
Phone: 011-44-844-261188

Americas

Brazil, Sao Paulo
Nishicom
Phone: 011-55-11-5351755



Publication No.: UG85
Stock No.: 050085-001
Revision No.: 1.0