# INTEL 80386

## PROGRAMMER'S REFERENCE MANUAL 1986



Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Distribution
Mail Stop SC6-59
3065 Bowers Avenue
Santa Clara, CA 95051

 CG-5/26/87 Edited 2001-02-01 by G.N.

# Customer Support

---

Customer Support is Intel's complete support service that provides Intel customers with hardware support, software support, customer training, and consulting services. For more information contact your local sales offices.

After a customer purchases any system hardware or software product, service and support become major factors in determining whether that product will continue to meet a customer's expectations. Such support requires an international support organization and a breadth of programs to meet a variety of customer needs. As you might expect, Intel's customer support is quite extensive. It includes factory repair services and worldwide field service offices providing hardware repair services, software support services, customer training classes, and consulting services.

Hardware Support Services

Intel is committed to providing an international service support package through a wide variety of service offerings available from Intel Hardware Support.

Software Support Services

Intel's software support consists of two levels of contracts. Standard support includes TIPS (Technical Information Phone Service), updates and subscription service (product-specific troubleshooting guides and COMMENTS Magazine). Basic support includes updates and the subscription service. Contracts are sold in environments which represent product groupings (i.e., iRMX environment).

Consulting Services

Intel provides field systems engineering services for any phase of your development or support effort. You can use our systems engineers in a variety of ways ranging from assistance in using a new product, developing an application, personalizing training, and customizing or tailoring an Intel product to providing technical and management consulting. Systems Engineers are well versed in technical areas such as microcommunications, real-time applications, embedded microcontrollers, and network services. You know your application needs; we know our products. Working together we can help you get a successful product to market in the least possible time.

Customer Training

Intel offers a wide range of instructional programs covering various aspects of system design and implementation. In just three to ten days a limited number of individuals learn more in a single workshop than in weeks of self-study. For optimum convenience, workshops are scheduled regularly at Training Centers woridwide or we can take our workshops to you for on-site instruction. Covering a wide variety of topics, Intel's major course categories include: architecture and assembly language, programming and operating systems, bitbus and LAN applications.

Training Center Locations

To obtain a complete catalog of our workshops, call the nearest Training Center in your area.

| | |
|---|---|
| Boston | (617) 692-1000 |
| Chicago | (312) 310-5700 |
| San Francisco | (415) 940-7800 |
| Washington D.C. | (301) 474-2878 |
| Isreal | (972) 349-491-099 |
| Tokyo | 03-437-6611 |
| Osaka (Call Tokyo) | 03-437-6611 |
| Toronto, Canada | (416) 675-2105 |
| London | (0793) 696-000 |
| Munich | (089) 5389-1 |
| Paris | (01) 687-22-21 |
| Stockholm | (468) 734-01-00 |
| Milan | 39-2-82-44-071 |
| Benelux (Rotterdam) | (10) 21-23-77 |
| Copenhagen | (1) 198-033 |
| Hong Kong | 5-215311-7 |

## Table of Contents

## Figures

## Tables

# Chapter 1 Introduction to the 80386

The 80386 is an advanced 32-bit microprocessor optimized for multitasking operating systems and designed for applications needing very high performance. The 32-bit registers and data paths support 32-bit addresses and data types. The processor can address up to four gigabytes of physical memory and 64 terabytes ($2^{46}$ bytes) of virtual memory. The on-chip memory-management facilities include address translation registers, advanced multitasking hardware, a protection mechanism, and paged virtual memory. Special debugging registers provide data and code breakpoints even in ROM-based software.

## 1.1 Organization of This Manual

This book presents the architecture of the 80386 in five parts:

```
Part I      —— Applications Programming
Part II     —— Systems Programming
Part III    —— Compatibility
Part IV     —— Instruction Set
Appendices
```

These divisions are determined in part by the architecture itself and in part by the different ways the book will be used. As the following table indicates, the latter two parts are intended as reference material for programmers actually engaged in the process of developing software for the 80386. The first three parts are explanatory, showing the purpose of architectural features, developing terminology and concepts, and describing instructions as they relate to specific purposes or to specific architectural features.

| | |
|---|---|
| Explanation | Part I —— Applications Programming<br>Part II —— Systems Programming<br>Part III —— Compatibility |
| Reference | Part IV —— Instruction Set<br>Appendices |

The first three parts follow the execution modes and protection features of the 80386 CPU. The distinction between applications features and systems features is determined by the protection mechanism of the 80386. One purpose of protection is to prevent applications from interfering with the operating system; therefore, the processor makes certain registers and instructions inaccessible to applications programs. The features discussed in Part I are those that are accessible to applications; the features in Part II are available only to systems software that has been given special privileges or in unprotected systems.

The processing mode of the 80386 also determines the features that are accessible. The 80386 has three processing modes:

1. Protected Mode.
2. Real-Address Mode.
3. Virtual 8086 Mode.

Protected mode is the natural 32-bit environment of the 80386 processor. In this mode all instructions and features are available.

Real-address mode (often called just "real mode") is the mode of the processor immediately after RESET. In real mode the 80386 appears to programmers as a fast 8086 with some new instructions. Most applications of the 80386 will use real mode for initialization only.

Virtual 8086 mode (also called V86 mode) is a dynamic mode in the sense that the processor can switch repeatedly and rapidly between V86 mode and protected mode. The CPU enters V86 mode from protected mode to execute an 8086 program, then leaves V86 mode and enters protected mode to continue executing a native 80386 program.

The features that are available to applications programs in protected mode and to all programs in V86 mode are the same. These features form the content of Part I. The additional features that are available to systems software in protected mode form Part II. Part III explains real-address mode and V86 mode, as well as how to execute a mix of 32-bit and 16-bit programs.

| | |
|---|---|
| Available in All Modes | Part I —— Applications Programming |
| Available in Protected Mode Only | Part II —— Systems Programming |
| Compatibility Modes | Part III —— Compatibility |

### 1.1.1 Part I —— Applications Programming

This part presents those aspects of the architecture that are customarily used by applications programmers.

Chapter 2 —— Basic Programming Model: Introduces the models of memory organization. Defines the data types. Presents the register set used by applications. Introduces the stack. Explains string operations. Defines the parts of an instruction. Explains addressing calculations. Introduces interrupts and exceptions as they may apply to applications programming.

Chapter 3 —— Application Instruction Set: Surveys the instructions commonly used for applications programming. Considers instructions in functionally related groups; for example, string instructions are considered in one section, while control-transfer instructions are considered in another. Explains the concepts behind the instructions. Details of individual instructions are deferred until Part IV, the instruction-set reference.

### 1.1.2 Part II —— Systems Programming

This part presents those aspects of the architecture that are customarily used by programmers who write operating systems, device drivers, debuggers, and other software that supports applications programs in the protected mode of the 80386.

Chapter 4 —— Systems Architecture: Surveys the features of the 80386 that are used by systems programmers. Introduces the remaining registers and data structures of the 80386 that were not discussed in Part I. Introduces the systems-oriented instructions in the context of the registers and data structures they support. Points to the chapter where each register, data structure, and instruction is considered in more detail.

Chapter 5 —— Memory Management: Presents details of the data structures, registers, and instructions that support virtual memory and the concepts of segmentation and paging. Explains how systems designers can choose a model of memory organization ranging from completely linear ("flat") to fully paged and segmented.

Chapter 6 —— Protection: Expands on the memory management features of the 80386 to include protection as it applies to both segments and pages. Explains the implementation of privilege rules, stack switching, pointer validation, user and supervisor modes. Protection aspects of multitasking are deferred until the following chapter.

Chapter 7 —— Multitasking: Explains how the hardware of the 80386 supports multitasking with context-switching operations and intertask protection.

Chapter 8 —— Input/Output: Reveals the I/O features of the 80386, including I/O instructions, protection as it relates to I/O, and the I/O permission map.

Chapter 9 —— Exceptions and Interrupts: Explains the basic interrupt mechanisms of the 80386. Shows how interrupts and exceptions relate to protection. Discusses all possible exceptions, listing causes and including information needed to handle and recover from the exception.

Chapter 10 —— Initialization: Defines the condition of the processor after RESET or power-up. Explains how to set up registers, flags, and data structures for either real-address mode or protected mode. Contains an example of an initialization program.

Chapter 11 —— Coprocessing and Multiprocessing: Explains the instructions and flags that support a numerics coprocessor and multiple CPUs with shared memory.

Chapter 12 —— Debugging: Tells how to use the debugging registers of the 80386.

### 1.1.3 Part III —— Compatibility

Other parts of the book treat the processor primarily as a 32-bit machine, omitting for simplicity its facilities for 16-bit operations. Indeed, the 80386 is a 32-bit machine, but its design fully supports 16-bit operands and addressing, too. This part completes the picture of the 80386 by explaining the features of the architecture that support 16-bit programs and 16-bit operations in 32-bit programs. All three processor modes are used to execute 16-bit programs: protected mode can directly execute 16-bit 80286 protected mode programs, real mode executes 8086 programs and real-mode 80286 programs, and virtual 8086 mode executes 8086 programs in a multitasking environment with other 80386 protected-mode programs. In addition, 32-bit and 16-bit modules and individual 32-bit and 16-bit operations can be mixed in protected mode.

Chapter 13 —— Executing 80286 Protected-Mode Code: In its protected mode, the 80386 can execute complete 80286 protected-mode systems, because 80286 capabilities are a subset of 80386 capabilities.

Chapter 14 —— 80386 Real-Address Mode: Explains the real mode of the 80386 CPU. In this mode the 80386 appears as a fast real-mode 80286 or fast 8086 enhanced with additional instructions.

Chapter 15 —— Virtual 8086 Mode: The 80386 can switch rapidly between its protected mode and V86 mode, giving it the ability to multiprogram 8086 programs along with "native mode" 32-bit programs.

Chapter 16 —— Mixing 16-Bit and 32-Bit Code: Even within a program or task, the 80386 can mix 16-bit and 32-bit modules. Furthermore, any given module can utilize both 16-bit and 32-bit operands and addresses.

### 1.1.4 Part IV —— Instruction Set

Parts I, II, and III present overviews of the instructions as they relate to specific aspects of the architecture, but this part presents the instructions in alphabetical order, providing the detail needed by assembly-language programmers and programmers of debuggers, compilers, operating systems, etc. Instruction descriptions include algorithmic description of operation, effect of flag settings, effect on flag settings, effect of operand- or address-size attributes, effect of processor modes, and possible exceptions.

### 1.1.5 Appendices

The appendices present tables of encodings and other details in a format designed for quick reference by assembly-language and systems programmers.

## 1.2 Related Literature

The following books contain additional material concerning the 80386 microprocessor:

- Introduction to the 80386, order number 231252
- 80386 Hardware Reference Manual, order number 231732
- 80386 System Software Writer's Guide, order number 231499
- 80386 High Performance 32-bit Microprocessor with Integrated Memory Management (Data Sheet), order number 231630

## 1.3 Notational Conventions

This manual uses special notations for data-structure formats, for symbolic representation of instructions, for hexadecimal numbers, and for super- and sub-scripts. Subscript characters are surrounded by {curly brackets}, for example 10{2} = 10 base 2. Superscript characters are preceeded by a caret and enclosed within (parentheses), for example 10^(3) = 10 to the third power. A review of these notations will make it easier to read the manual.

### 1.3.1 Data-Structure Formats

In illustrations of data structures in memory, smaller addresses appear at the lower-right part of the figure; addresses increase toward the left and upwards. Bit positions are numbered from right to left. Figure 1-1 illustrates this convention.

### 1.3.2 Undefined Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as undefined. When bits are marked as undefined (as illustrated in Figure 1-1), it is essential for compatibility with future processors that software treat these bits as undefined. Software should follow these guidelines in dealing with undefined bits:

- Do not depend on the states of any undefined bits when testing the values of registers that contain such bits. Mask out the undefined bits before testing.
- Do not depend on the states of any undefined bits when storing them in memory or in another register.
- Do not depend on the ability to retain information written into any undefined bits.

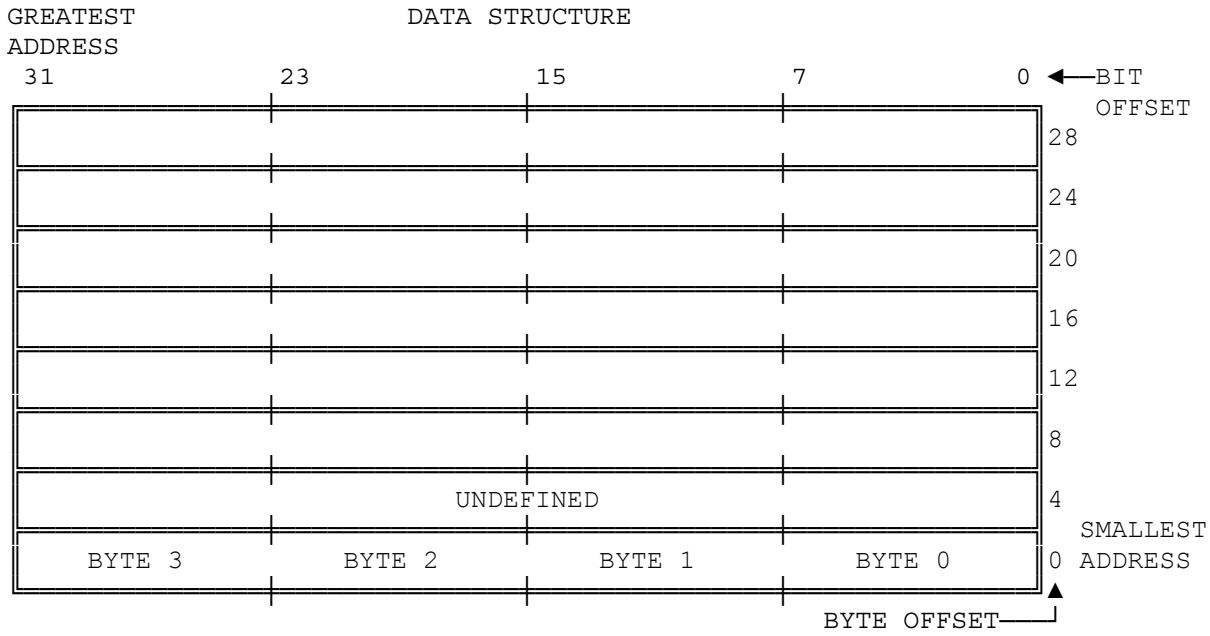
- When loading a register, always load the undefined bits as zeros or reload them with values previously stored from the same register.

---

NOTE
Depending upon the values of undefined register bits will make software dependent upon the unspecified manner in which the 80386 handles these bits. Depending upon undefined values risks making software incompatible with future processors that define usages for these bits. AVOID ANY SOFTWARE DEPENDENCE UPON THE STATE OF UNDEFINED 80386 REGISTER BITS.

---

**Figure 1-1. Example Data Structure**




### 1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of the 80386 Assembly Language is used. In this subset, an instruction has the following format:

label: prefix mnemonic argument1, argument2, argument3

where:

- A label is an identifier that is followed by a colon.
- A prefix is an optional reserved name for one of the instruction prefixes.
- A mnemonic is a reserved name for a class of instruction opcodes that have the same function.

- The operands argument1, argument2, and argument3 are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example). When two operands are present in an instruction that modifies data, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand.

### 1.3.4 Hexadecimal Numbers

Base 16 numbers are represented by a string of hexadecimal digits followed by the character H. A hexadecimal digit is a character from the set (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). In some cases, especially in examples of program syntax, a leading zero is added if the number would otherwise begin with one of the digits A-F. For example, 0FH is equivalent to the decimal number 15.

### 1.3.5 Sub- and Super-Scripts

This manual uses special notation to represent sub- and super-script characters. Sub-script characters are surrounded by {curly brackets}, for example 10{2} = 10 base 2. Super-script characters are preceeded by a caret and enclosed within (parentheses), for example 10^(3) = 10 to the third power.

Editors Note: This revised document provides actual super-script and sub-script characters where appropriate.

# PART I APPLICATIONS PROGRAMMING

# Chapter 2 Basic Programming Model

---

This chapter describes the 80386 application programming environment as seen by assembly language programmers when the processor is executing in protected mode. The chapter introduces programmers to those features of the 80386 architecture that directly affect the design and implementation of 80386 applications programs. Other chapters discuss 80386 features that relate to systems programming or to compatibility with other processors of the 8086 family.

The basic programming model consists of these aspects:

- Memory organization and segmentation
- Data types
- Registers
- Instruction format
- Operand selection
- Interrupts and exceptions

Note that input/output is not included as part of the basic programming model. Systems designers may choose to make I/O instructions available to applications or may choose to reserve these functions for the operating system. For this reason, the I/O features of the 80386 are discussed in Part II.

This chapter contains a section for each aspect of the architecture that is normally visible to applications.

## 2.1 Memory Organization and Segmentation

The physical memory of an 80386 system is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address that ranges from zero to a maximum of $2^{32}$-1 (4 gigabytes).

80386 programs, however, are independent of the physical address space. This means that programs can be written without knowledge of how much physical memory is available and without knowledge of exactly where in physical memory the instructions and data are located.

The model of memory organization seen by applications programmers is determined by systems-software designers. The architecture of the 80386 gives designers the freedom to choose a model for each task. The model of memory organization can range between the following extremes:

- A "flat" address space consisting of a single array of up to 4 gigabytes.

- A segmented address space consisting of a collection of up to 16,383 linear address spaces of up to 4 gigabytes each.

Both models can provide memory protection. Different tasks may employ different models of memory organization. The criteria that designers use to determine a memory organization model and the means that systems programmers use to implement that model are covered in Part II——Systems Programming.

### 2.1.1 The "Flat" Model

In a "flat" model of memory organization, the applications programmer sees a single array of up to $2^{32}$ bytes (4 gigabytes). While the physical memory can contain up to 4 gigabytes, it is usually much smaller; the processor maps the 4 gigabyte flat space onto the physical address space by the address translation mechanisms described in Chapter 5. Applications programmers do not need to know the details of the mapping.

A pointer into this flat address space is a 32-bit ordinal number that may range from 0 to $2^{32}$-1. Relocation of separately-compiled modules in this space must be performed by systems software (e.g., linkers, locators, binders, loaders).

### 2.1.2 The Segmented Model

In a segmented model of memory organization, the address space as viewed by an applications program (called the logical address space) is a much larger space of up to $2^{46}$ bytes (64 terabytes). The processor maps the 64 terabyte logical address space onto the physical address space (up to 4 gigabytes) by the address translation mechanisms described in Chapter 5. Applications programmers do not need to know the details of this mapping.

Applications programmers view the logical address space of the 80386 as a collection of up to 16,383 one-dimensional subspaces, each with a specified length. Each of these linear subspaces is called a segment. A segment is a unit of contiguous address space. Segment sizes may range from one byte up to a maximum of $2^{32}$ bytes (4 gigabytes).

A complete pointer in this address space consists of two parts (see Figure 2-1):

1. A segment selector, which is a 16-bit field that identifies a segment.

2. An offset, which is a 32-bit ordinal that addresses to the byte level within a segment.

During execution of a program, the processor associates with a segment selector the physical address of the beginning of the segment. Separately compiled modules can be relocated at run time by changing the base address of their segments. The size of a segment is variable; therefore, a segment can be exactly the size of the module it contains.

## 2.2 Data Types

Bytes, words, and doublewords are the fundamental data types (refer to Figure 2-2). A byte is eight contiguous bits starting at any logical address. The bits are numbered 0 through 7; bit zero is the least significant bit.
A word is two contiguous bytes starting at any byte address. A word thus contains 16 bits. The bits of a word are numbered from 0 through 15; bit 0 is the least significant bit. The byte containing bit 0 of the word is called the low byte; the byte containing bit 15 is called the high byte.

Each byte within a word has its own address, and the smaller of the addresses is the address of the word. The byte at this lower address contains the eight least significant bits of the word, while the byte at the higher address contains the eight most significant bits.

A doubleword is two contiguous words starting at any byte address. A doubleword thus contains 32 bits. The bits of a doubleword are numbered from 0 through 31; bit 0 is the least significant bit. The word containing bit 0 of the doubleword is called the low word; the word containing bit 31 is called the high word.

Each byte within a doubleword has its own address, and the smallest of the addresses is the address of the doubleword. The byte at this lowest address contains the eight least significant bits of the doubleword, while the byte at the highest address contains the eight most significant bits. Figure 2-3 illustrates the arrangement of bytes within words anddoublewords.

Note that words need not be aligned at even-numbered addresses and doublewords need not be aligned at addresses evenly divisible by four. This allows maximum flexibility in data structures (e.g., records containing mixed byte, word, and doubleword items) and efficiency in memory utilization. When used in a configuration with a 32-bit bus, actual transfers of data between processor and memory take place in units of doublewords beginning at addresses evenly divisible by four; however, the processor converts requests for misaligned words or doublewords into the appropriate sequences of requests acceptable to the memory interface. Such misaligned data transfers reduce performance by requiring extra memory cycles. For maximum performance, data structures (including stacks) should be designed in such a way that, whenever possible, word operands are aligned at even addresses and doubleword operands are aligned at addresses evenly divisible by four. Due to instruction prefetching and queuing within the CPU, there is no requirement for instructions to be aligned on word or doubleword boundaries. (However, a slight increase in speed results if the target addresses of control transfers are evenly divisible by four.)

Although bytes, words, and doublewords are the fundamental types of operands, the processor also supports additional interpretations of these operands. Depending on the instruction referring to the operand, the following additional data types are recognized:

Integer:
A signed binary numeric value contained in a 32-bit doubleword,16-bit word, or 8-bit byte. All operations assume a 2's complement representation. The sign bit is located in bit 7 in a byte, bit 15 in a word, and bit 31 in a doubleword. The sign bit has the value zero for positive integers and one

for negative. Since the high-order bit is used for a sign, the range of an 8-bit integer is -128 through +127; 16-bit integers may range from -32,768 through +32,767; 32-bit integers may range from $-2^{31}$ through $+2^{31}-1$. The value zero has a positive sign.

Ordinal:
An unsigned binary numeric value contained in a 32-bit doubleword, 16-bit word, or 8-bit byte. All bits are considered in determining magnitude of the number. The value range of an 8-bit ordinal number is 0-255; 16 bits can represent values from 0 through 65,535; 32 bits can represent values from 0 through $2^{32}-1$.

Near Pointer:
A 32-bit logical address. A near pointer is an offset within a segment. Near pointers are used in either a flat or a segmented model of memory organization.

Far Pointer:
A 48-bit logical address of two components: a 16-bit segment selector component and a 32-bit offset component. Far pointers are used by applications programmers only when systems designers choose a segmented memory organization.

String:
A contiguous sequence of bytes, words, or doublewords. A string may contain from zero bytes to $2^{32}-1$ bytes (4 gigabytes).

Bit field:
A contiguous sequence of bits. A bit field may begin at any bit position of any byte and may contain up to 32 bits.

Bit string:
A contiguous sequence of bits. A bit string may begin at any bit position of any byte and may contain up to $2^{32}-1$ bits.

BCD:
A byte (unpacked) representation of a decimal digit in the range0 through 9. Unpacked decimal numbers are stored as unsigned byte quantities. One digit is stored in each byte. The magnitude of the number is determined from the low-order half-byte; hexadecimal values 0-9 are valid and are interpreted as decimal numbers. The high-order half-byte must be zero for multiplication and division; it may contain any value for addition and subtraction.

Packed BCD:
A byte (packed) representation of two decimal digits, each in the range 0 through 9. One digit is stored in each half-byte. The digit in the high-order half-byte is the most significant. Values 0-9 are valid in each half-byte. The range of a packed decimal byte is 0-99.

Figure 2-4 graphically summarizes the data types supported by the 80386.

**Figure 2-1. Two-Component Pointer**

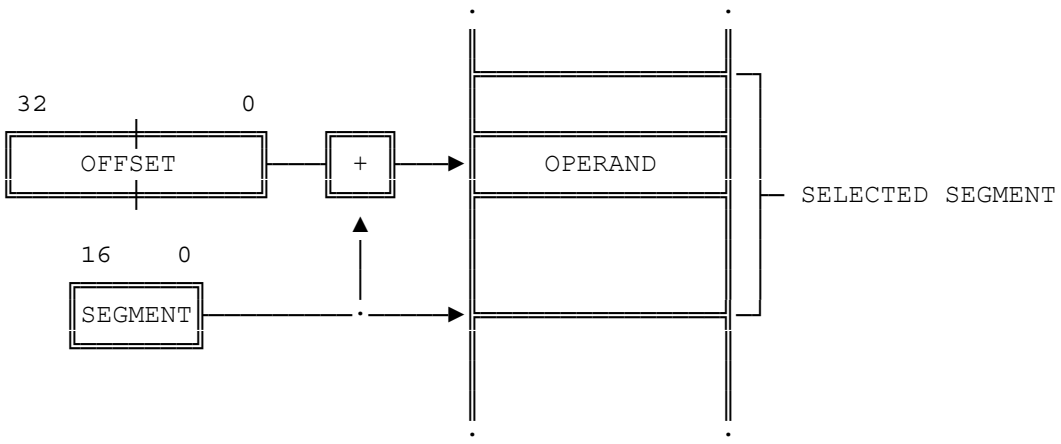



**Figure 2-2. Fundamental Data Types**

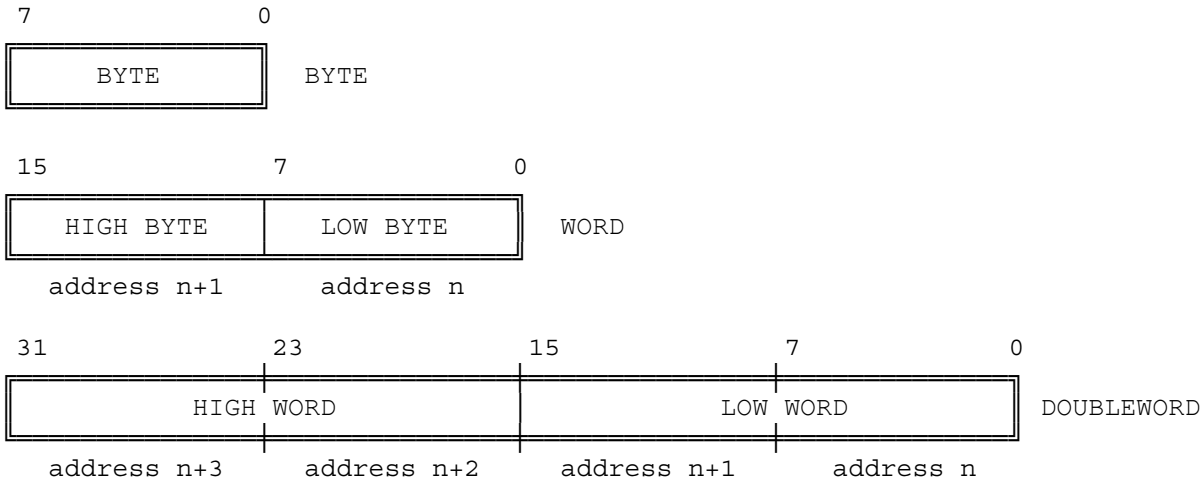


**Figure 2-3. Bytes, Words, and Doublewords in Memory**

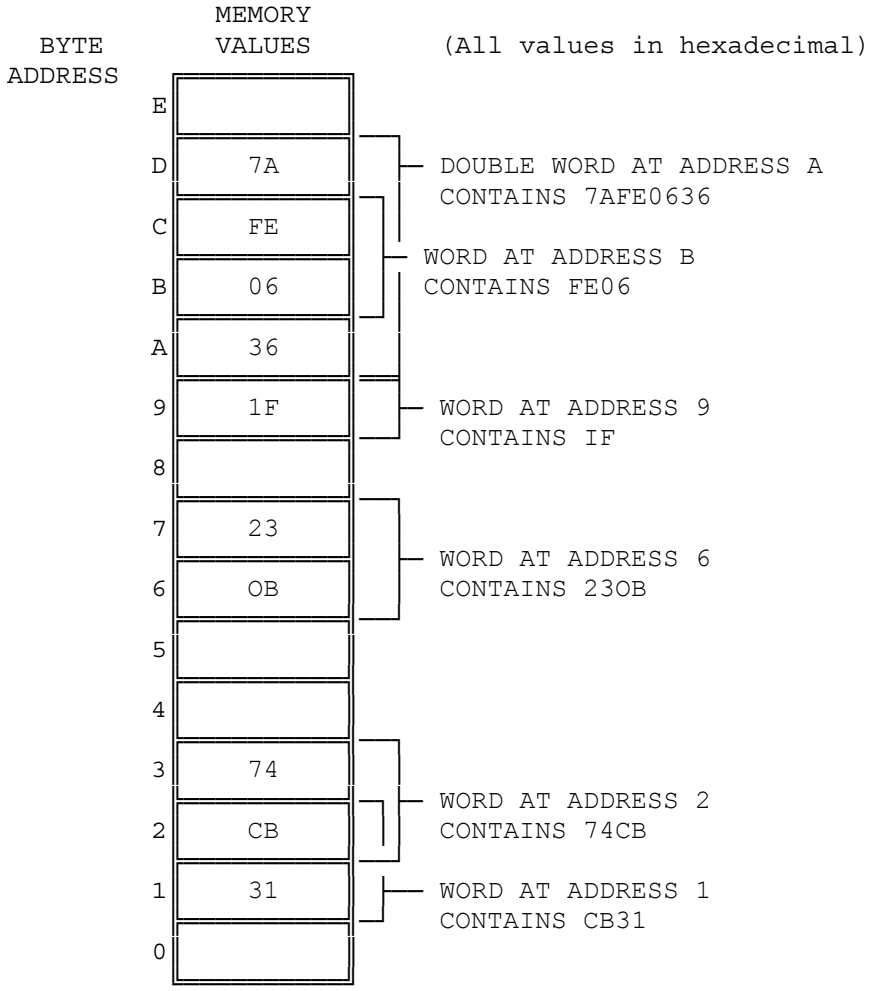
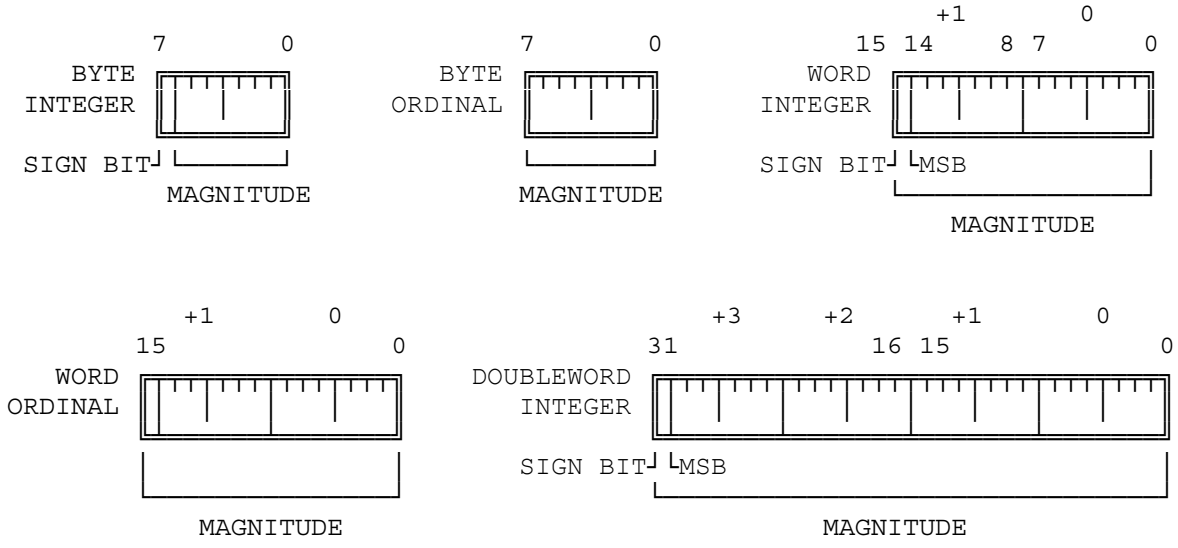
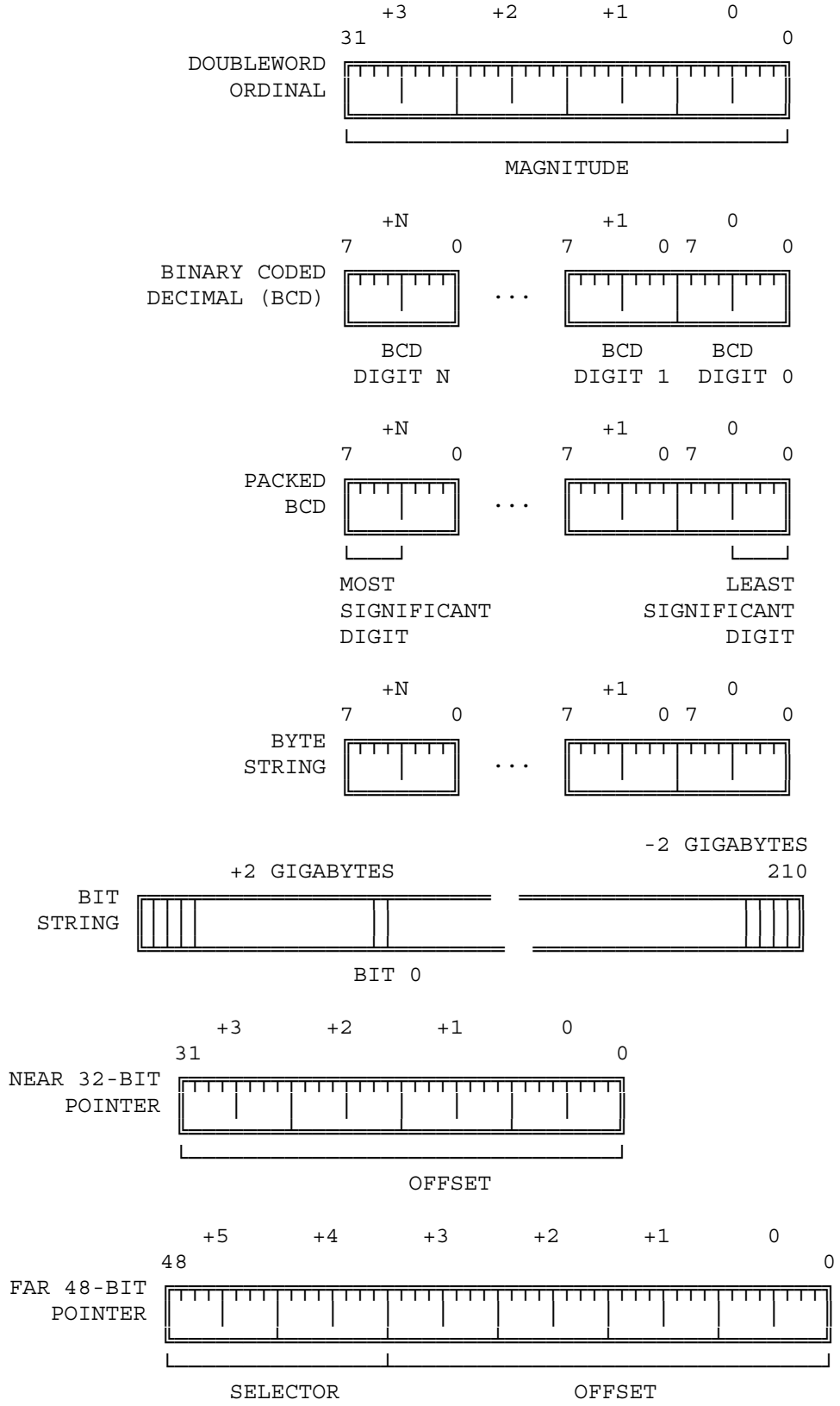
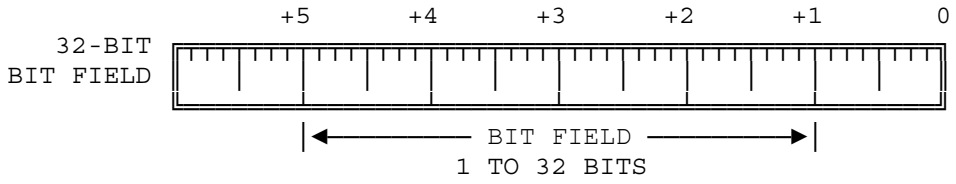



**Figure 2-4. 80386 Data Types**




+3 +2 +1 0
31 0
DOUBLEWORD ORDINAL
MAGNITUDE
+N +1 0
7 0 7 0 7 0
BINARY CODED DECIMAL (BCD)
BCD DIGIT N
BCD DIGIT 1
BCD DIGIT 0
PACKED BCD
MOST SIGNIFICANT DIGIT
LEAST SIGNIFICANT DIGIT
BYTE STRING
-2 GIGABYTES
+2 GIGABYTES
210
BIT STRING
BIT 0
NEAR 32-BIT POINTER
OFFSET
+5 +4 +3 +2 +1 0
48 0
FAR 48-BIT POINTER
SELECTOR
OFFSET




## 2.3 Registers

The 80386 contains a total of sixteen registers that are of interest to the applications programmer. As Figure 2-5 shows, these registers may be grouped into these basic categories:

1. General registers. These eight 32-bit general-purpose registers are used primarily to contain operands for arithmetic and logical operations.

2. Segment registers. These special-purpose registers permit systems software designers to choose either a flat or segmented model of memory organization. These six registers determine, at any given time, which segments of memory are currently addressable.

3. Status and instruction registers. These special-purpose registers are used to record and alter certain aspects of the 80386 processor state.

### 2.3.1 General Registers

The general registers of the 80386 are the 32-bit registers EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. These registers are used interchangeably to contain the operands of logical and arithmetic operations. They may also be used interchangeably for operands of address computations (except that ESP cannot be used as an index operand).

As Figure 2-5 shows, the low-order word of each of these eight registers has a separate name and can be treated as a unit. This feature is useful for handling 16-bit data items and for compatibility with the 8086 and 80286 processors. The word registers are named AX, BX, CX, DX, BP, SP, SI, and DI.

Figure 2-5 also illustrates that each byte of the 16-bit registers AX, BX, CX, and DX has a separate name and can be treated as a unit. This feature is useful for handling characters and other 8-bit data items. The byte registers are named AH, BH, CH, and DH (high bytes); and AL, BL, CL, and DL (low bytes).

All of the general-purpose registers are available for addressing calculations and for the results of most arithmetic and logical calculations; however, a few functions are dedicated to certain registers. By implicitly choosing registers for these functions, the 80386 architecture can encode instructions more compactly. The instructions that use specific registers include: double-precision multiply and divide, I/O, string instructions, translate, loop, variable shift and rotate, and stack operations.

### 2.3.2 Segment Registers

The segment registers of the 80386 give systems software designers the flexibility to choose among various models of memory organization. Implementation of memory models is the subject of Part II —— Systems Programming. Designers may choose a model in which applications programs do not need to modify segment registers, in which case applications programmers may skip this section.

Complete programs generally consist of many different modules, each consisting of instructions and data. However, at any given time during program execution, only a small subset of a program's modules are actually in use. The 80386 architecture takes advantage of this by providing mechanisms to support direct access to the instructions and data of the current module's environment, with access to additional segments on demand.

At any given instant, six segments of memory may be immediately accessible to an executing 80386 program. The segment registers CS, DS, SS, ES, FS, and GS are used to identify these six current segments. Each of these registers specifies a particular kind of segment, as characterized by the associated mnemonics ("code," "data," or "stack") shown in Figure 2-6. Each register uniquely determines one particular segment, from among the segments that make up the program, that is to be immediately accessible at highest speed.

The segment containing the currently executing sequence of instructions is known as the current code segment; it is specified by means of the CS register. The 80386 fetches all instructions from this code segment, using as an offset the contents of the instruction pointer. CS is changed implicitly as the result of intersegment control-transfer instructions (for example, CALL and JMP), interrupts, and exceptions.

Subroutine calls, parameters, and procedure activation records usually require that a region of memory be allocated for a stack. All stack operations use the SS register to locate the stack. Unlike CS, the SS register can be loaded explicitly, thereby permitting programmers to define stacks dynamically.

The DS, ES, FS, and GS registers allow the specification of four data segments, each addressable by the currently executing program. Accessibility to four separate data areas helps programs efficiently access different types of data structures; for example, one data segment register can point to the data structures of the current module, another to the exported data of a higher-level module, another to a dynamically created data structure, and another to data shared with another task. An operand within a data segment is addressed by specifying its offset either directly in an instruction or indirectly via general registers.

Depending on the structure of data (e.g., the way data is parceled into one or more segments), a program may require access to more than four data segments. To access additional segments, the DS, ES, FS, and GS registers can be changed under program control during the course of a program's execution. This simply requires that the program execute an instruction to load the appropriate segment register prior to executing instructions that access the data.

The processor associates a base address with each segment selected by a segment register. To address an element within a segment, a 32-bit offset is added to the segment's base address. Once a segment is selected (by loading the segment selector into a segment register), a data manipulation instruction only needs to specify the offset. Simple rules define which segment register is used to form an address when only an offset is specified.

**Figure 2-5. 80386 Applications Register Set**

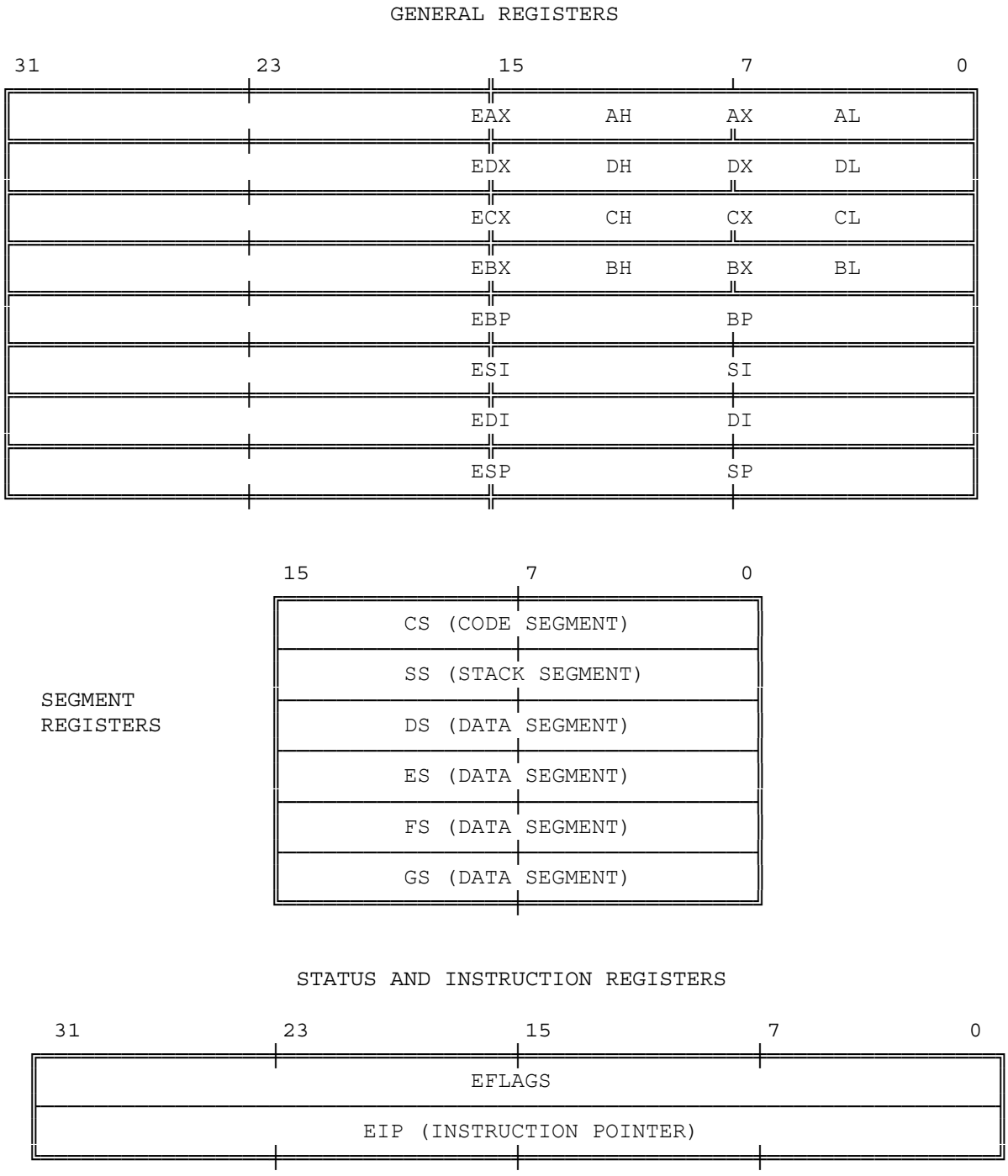
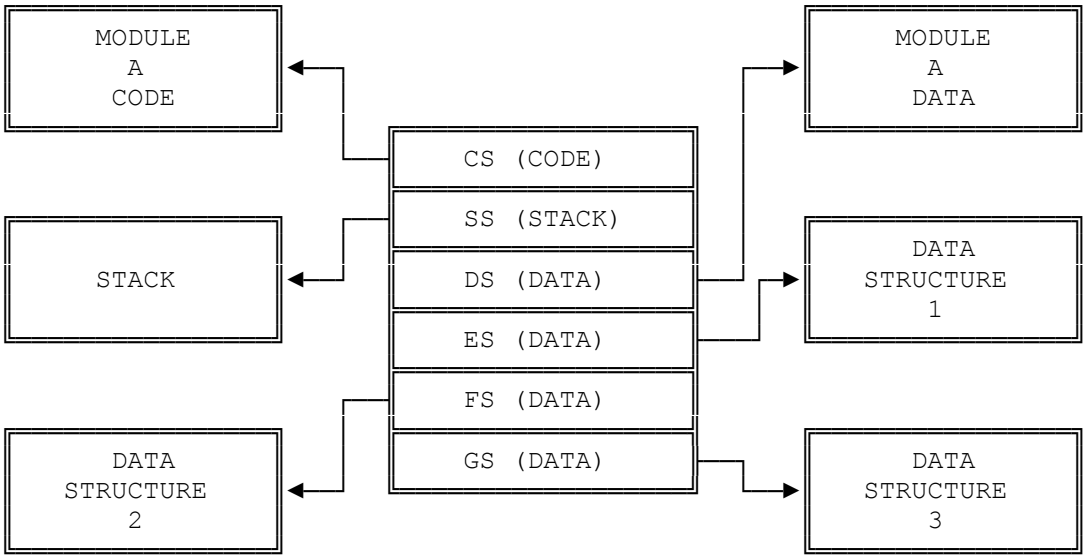


**Figure 2-6. Use of Memory Segmentation**




### 2.3.3 Stack Implementation

Stack operations are facilitated by three registers:

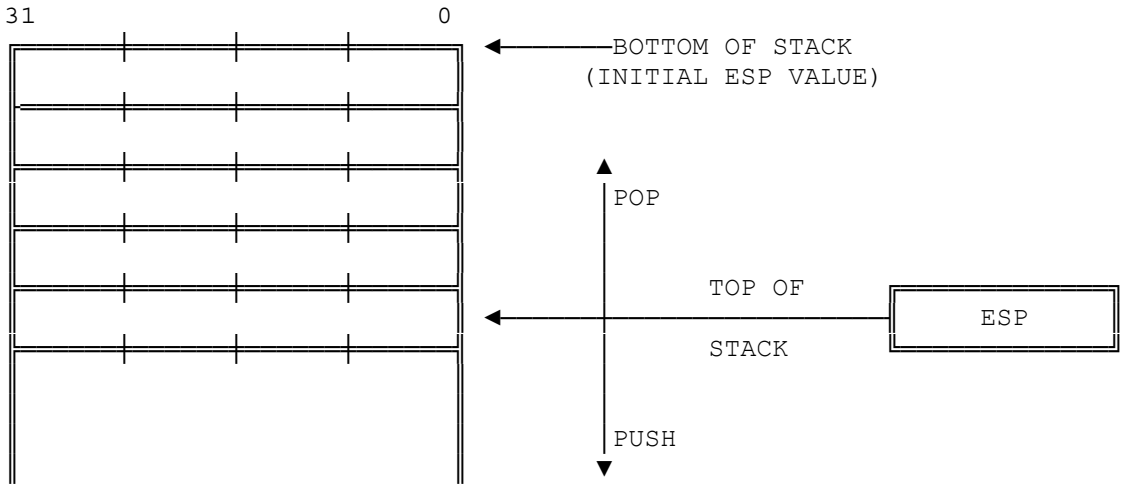
1. The stack segment (SS) register. Stacks are implemented in memory. A system may have a number of stacks that is limited only by the maximum number of segments. A stack may be up to 4 gigabytes long, the maximum length of a segment. One stack is directly addressable at a time——the one located by SS. This is the current stack, often referred to simply as "the" stack. SS is used automatically by the processor for all stack operations.

2. The stack pointer (ESP) register. ESP points to the top of the push-down stack (TOS). It is referenced implicitly by PUSH and POP operations, subroutine calls and returns, and interrupt operations. When an item is pushed onto the stack (see Figure 2-7), the processor decrements ESP, then writes the item at the new TOS. When an item is popped off the stack, the processor copies it from TOS, then increments ESP. In other words, the stack grows down in memory toward lesser addresses.

3. The stack-frame base pointer (EBP) register. The EBP is the best choice of register for accessing data structures, variables and dynamically allocated work space within the stack. EBP is often used to access elements on the stack relative to a fixed point on the stack rather than relative to the current TOS. It typically identifies the base address of the current stack frame established for the current procedure. When EBP is used as the base register in an offset calculation, the offset is calculated automatically in the current stack segment (i.e., the segment currently selected by SS). Because

SS does not have to be explicitly specified, instruction encoding in such cases is more efficient. EBP can also be used to index into segments addressable via other segment registers.

**Figure 2-7. 80386 Stack**



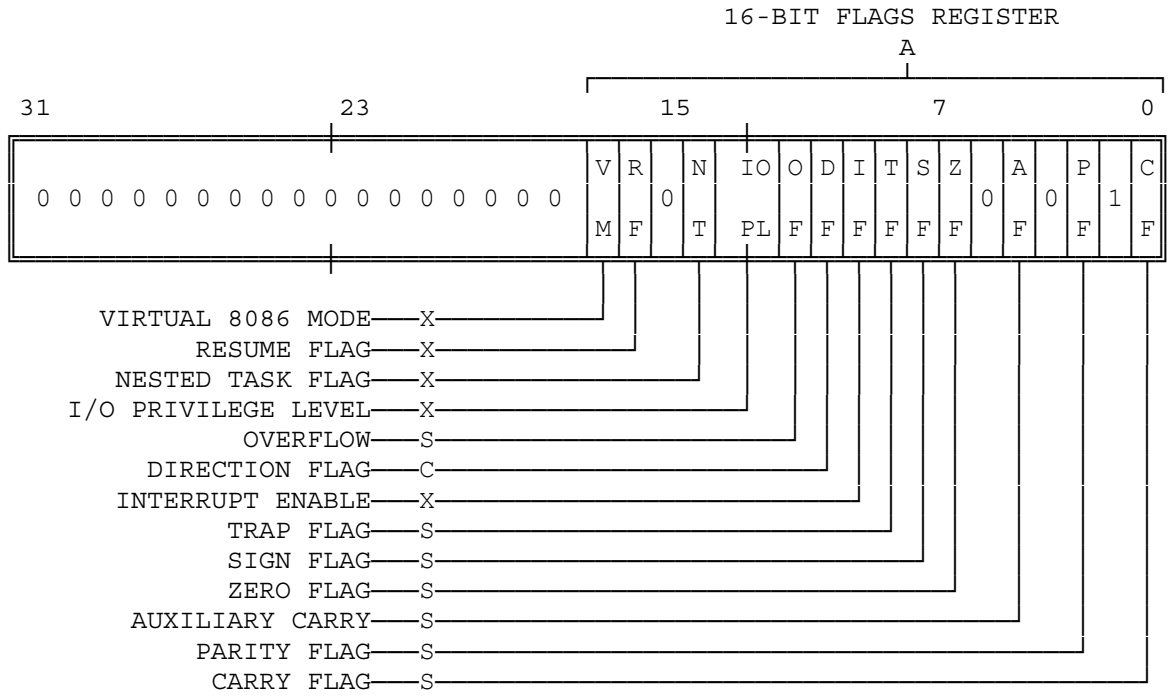

## 2.3.4 Flags Register

The flags register is a 32-bit register named EFLAGS. Figure 2-8 defines the bits within this register. The flags control certain operations and indicate the status of the 80386.

The low-order 16 bits of EFLAGS is named FLAGS and can be treated as a unit. This feature is useful when executing 8086 and 80286 code, because this part of EFLAGS is identical to the FLAGS register of the 8086 and the 80286.

The flags may be considered in three groups: the status flags, the control flags, and the systems flags. Discussion of the systems flags is delayed until Part II.

**Figure 2-8. EFLAGS Register**




S = STATUS FLAG, C = CONTROL FLAG, X = SYSTEM FLAG

NOTE: 0 OR 1 INDICATES INTEL RESERVED. DO NOT DEFINE

### 2.3.4.1 Status Flags

The status flags of the EFLAGS register allow the results of one instruction to influence later instructions. The arithmetic instructions use OF, SF, ZF, AF, PF, and CF. The SCAS (Scan String), CMPS (Compare String), and LOOP instructions use ZF to signal that their operations are complete. There are instructions to set, clear, and complement CF before execution of an arithmetic instruction. Refer to Appendix C for definition of each status flag.

### 2.3.4.2 Control Flag

The control flag DF of the EFLAGS register controls string instructions.

DF (Direction Flag, bit 10)

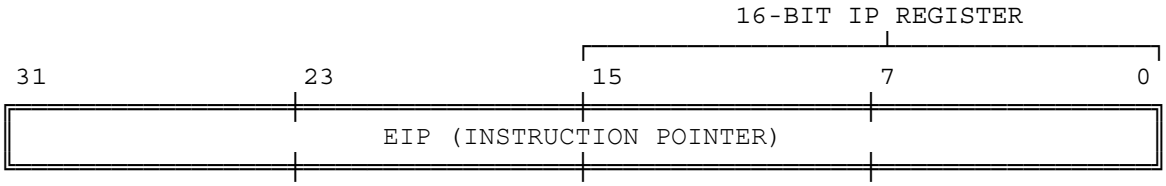
Setting DF causes string instructions to auto-decrement; that is, to process strings from high addresses to low addresses. Clearing DF causes string instructions to auto-increment, or to process strings from low addresses to high addresses.

#### 2.3.4.3 Instruction Pointer

The instruction pointer register (EIP) contains the offset address, relative to the start of the current code segment, of the next sequential instruction to be executed. The instruction pointer is not directly visible to the programmer; it is controlled implicitly by control-transfer instructions, interrupts, and exceptions.

As Figure 2-9 shows, the low-order 16 bits of EIP is named IP and can be used by the processor as a unit. This feature is useful when executing instructions designed for the 8086 and 80286 processors.

**Figure 2-9. Instruction Pointer Register**




## 2.4 Instruction Format

The information encoded in an 80386 instruction includes a specification of the operation to be performed, the type of the operands to be manipulated, and the location of these operands. If an operand is located in memory, the instruction must also select, explicitly or implicitly, which of the currently addressable segments contains the operand.

80386 instructions are composed of various elements and have various formats. The exact format of instructions is shown in Appendix B; the elements of instructions are described below. Of these instruction elements, only one, the opcode, is always present. The other elements may or may not be present, depending on the particular operation involved and on the location and type of the operands. The elements of an instruction, in order of occurrence are as follows:

- Prefixes —— one or more bytes preceding an instruction that modify the operation of the instruction. The following types of prefixes can be used by applications programs:

    1. Segment override —— explicitly specifies which segment register an instruction should use, thereby overriding the default segment-register selection used by the 80386 for that instruction.

    2. Address size —— switches between 32-bit and 16-bit address generation.

    3. Operand size —— switches between 32-bit and 16-bit operands.

    4. Repeat —— used with a string instruction to cause the instruction to act on each element of the string.

- Opcode —— specifies the operation performed by the instruction. Some operations have several different opcodes, each specifying a different variant of the operation.

- Register specifier —— an instruction may specify one or two register operands. Register specifiers may occur either in the same byte as the opcode or in the same byte as the addressing-mode specifier.

- Addressing-mode specifier —— when present, specifies whether an operand is a register or memory location; if in memory, specifies whether a displacement, a base register, an index register, and scaling are to be used.

- SIB (scale, index, base) byte —— when the addressing-mode specifier indicates that an index register will be used to compute the address of an operand, an SIB byte is included in the instruction to encode the base register, the index register, and a scaling factor.

- Displacement —— when the addressing-mode specifier indicates that a displacement will be used to compute the address of an operand, the displacement is encoded in the instruction. A displacement is a signed integer of 32, 16, or eight bits. The eight-bit form is used in the common case when the displacement is sufficiently small. The processor extends an eight-bit displacement to 16 or 32 bits, taking into account the sign.

- Immediate operand —— when present, directly provides the value of an operand of the instruction. Immediate operands may be 8, 16, or 32 bits wide. In cases where an eight-bit immediate operand is combined in some way with a 16- or 32-bit operand, the processor automatically extends the size of the eight-bit operand, taking into account the sign.

## 2.5 Operand Selection

An instruction can act on zero or more operands, which are the data manipulated by the instruction. An example of a zero-operand instruction is NOP (no operation). An operand can be in any of these locations:

- In the instruction itself (an immediate operand)

- In a register (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP in the case of 32-bit operands; AX, BX, CX, DX, SI, DI, SP, or BP in the case of 16-bit operands; AH, AL, BH, BL, CH, CL, DH, or DL in the case of 8-bit operands; the segment registers; or the EFLAGS register for flag operations)

- In memory

- At an I/O port

Immediate operands and operands in registers can be accessed more rapidly than operands in memory since memory operands must be fetched from memory. Register operands are available in the CPU. Immediate operands are also available in the CPU, because they are prefetched as part of the instruction.

Of the instructions that have operands, some specify operands implicitly; others specify operands explicitly; still others use a combination of implicit and explicit specification; for example:

Implicit operand: AAM

By definition, AAM (ASCII adjust for multiplication) operates on the contents of the AX register.

Explicit operand: XCHG EAX, EBX

The operands to be exchanged are encoded in the instruction after the opcode.

Implicit and explicit operands: PUSH COUNTER

The memory variable COUNTER (the explicit operand) is copied to the top of the stack (the implicit operand).

Note that most instructions have implicit operands. All arithmetic instructions, for example, update the EFLAGS register.

An 80386 instruction can explicitly reference one or two operands. Two-operand instructions, such as MOV, ADD, XOR, etc., generally overwrite one of the two participating operands with the result. A distinction can thus be made between the source operand (the one unaffected by the operation) and the destination operand (the one overwritten by the result).

For most instructions, one of the two explicitly specified operands——either the source or the destination——can be either in a register or in memory. The other operand must be in a register or be an immediate source operand. Thus, the explicit two-operand instructions of the 80386 permit operations of the following kinds:

- Register-to-register
- Register-to-memory
- Memory-to-register
- Immediate-to-register
- Immediate-to-memory

Certain string instructions and stack manipulation instructions, however, transfer data from memory to memory. Both operands of some string instructions are in memory and are implicitly specified. Push and pop stack operations allow transfer between memory operands and the memory-based stack.

### 2.5.1 Immediate Operands

Certain instructions use data from the instruction itself as one (and sometimes two) of the operands. Such an operand is called an immediate operand. The operand may be 32-, 16-, or 8-bits long. For example:

SHR PATTERN, 2

One byte of the instruction holds the value 2, the number of bits by which to shift the variable PATTERN.

```
TEST PATTERN, 0FFFF00FFH
```

A doubleword of the instruction holds the mask that is used to test the variable PATTERN.

### 2.5.2 Register Operands

Operands may be located in one of the 32-bit general registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP), in one of the 16-bit general registers (AX, BX, CX, DX, SI, DI, SP, or BP), or in one of the 8-bit general registers (AH, BH, CH, DH, AL, BL, CL,or DL).

The 80386 has instructions for referencing the segment registers (CS, DS, ES, SS, FS, GS). These instructions are used by applications programs only if systems designers have chosen a segmented memory model.

The 80386 also has instructions for referring to the flag register. The flags may be stored on the stack and restored from the stack. Certain instructions change the commonly modified flags directly in the EFLAGS register. Other flags that are seldom modified can be modified indirectly via the flags image in the stack.

### 2.5.3 Memory Operands

Data-manipulation instructions that address operands in memory must specify (either directly or indirectly) the segment that contains the operand and the offset of the operand within the segment. However, for speed and compact instruction encoding, segment selectors are stored in the high speed segment registers. Therefore, data-manipulation instructions need to specify only the desired segment register and an offset in order to address a memory operand.

An 80386 data-manipulation instruction that accesses memory uses one of the following methods for specifying the offset of a memory operand within its segment:

1. Most data-manipulation instructions that access memory contain a byte that explicitly specifies the addressing method for the operand. A byte, known as the modR/M byte, follows the opcode and specifies whether the operand is in a register or in memory. If the operand is in memory, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement. When an index register is used, the modR/M byte is also followed by another byte that identifies the index register and scaling factor. This addressing method is the mostflexible.

2. A few data-manipulation instructions implicitly use specialized addressing methods:

   - For a few short forms of MOV that implicitly use the EAX register, the offset of the operand is coded as a doubleword in the instruction. No base register, index register, or scaling factor are used.

   - String operations implicitly address memory via DS:ESI, (MOVS, CMPS, OUTS, LODS, SCAS) or via ES:EDI (MOVS, CMPS, INS, STOS).

   - Stack operations implicitly address operands via SS:ESP registers; e.g., PUSH, POP, PUSHA, PUSHAD, POPA, POPAD, PUSHF, PUSHFD, POPF, POPFD, CALL, RET, IRET, IRETD, exceptions, and interrupts.

#### 2.5.3.1 Segment Selection

Data-manipulation instructions need not explicitly specify which segment register is used. For all of these instructions, specification of a segment register is optional. For all memory accesses, if a segment is not explicitly specified by the instruction, the processor automatically chooses a segment register according to the rules of Table 2-1. (If systems designers have chosen a flat model of memory organization, the segment registers and the rules that the processor uses in choosing them are not apparent to applications programs.)

There is a close connection between the kind of memory reference and the segment in which that operand resides. As a rule, a memory reference implies the current data segment (i.e., the implicit segment selector is in DS). However, ESP and EBP are used to access items on the stack; therefore, when the ESP or EBP register is used as a base register, the current stack segment is implied (i.e., SS contains the selector).

Special instruction prefix elements may be used to override the default segment selection. Segment-override prefixes allow an explicit segment selection. The 80386 has a segment-override prefix for each of the segment registers. Only in the following special cases is there an implied segment selection that a segment prefix cannot override:

- The use of ES for destination strings in string instructions.
- The use of SS in stack instructions.
- The use of CS for instruction fetches.

**Table 2-1. Default Segment Register Selection Rules**

| Memory Reference Needed | Segment Register Used | Implicit Segment Selection Rule |
|---|---|---|
| Instructions | Code (CS) | Automatic with instruction prefetch |
| Stack | Stack (SS) | All stack pushes and pops. Any memory reference that uses ESP or EBP as a base register. |
| Local Data | Data (DS) | All data references except when relative to stack or string destination. |
| Destination Strings | Extra (ES) | Destination of string instructions. |

### 2.5.3.2 Effective-Address Computation

The modR/M byte provides the most flexible of the addressing methods, and instructions that require a modR/M byte as the second byte of the instruction are the most common in the 80386 instruction set. For memory operands defined by modR/M, the offset within the desired segment is calculated by taking the sum of up to three components:

- A displacement element in the instruction.
- A base register.
- An index register. The index register may be automatically multiplied by a scaling factor of 2, 4, or 8.

The offset that results from adding these components is called an effective address. Each of these components of an effective address may have either a positive or negative value. If the sum of all the components exceeds $2^{32}$, the effective address is truncated to 32 bits.Figure 2-10 illustrates the full set of possibilities for modR/M addressing.

The displacement component, because it is encoded in the instruction, is useful for fixed aspects of addressing; for example:

- Location of simple scalar operands.
- Beginning of a statically allocated array.
- Offset of an item within a record.

The base and index components have similar functions. Both utilize the same set of general registers. Both can be used for aspects of addressing that are determined dynamically; for example:

- Location of procedure parameters and local variables in stack.
- The beginning of one record among several occurrences of the same record type or in an array of records.
- The beginning of one dimension of multiple dimension array.

- The beginning of a dynamically allocated array.

The uses of general registers as base or index components differ in the following respects:

- ESP cannot be used as an index register.
- When ESP or EBP is used as the base register, the default segment is the one selected by SS. In all other cases the default segment is DS.

The scaling factor permits efficient indexing into an array in the common cases when array elements are 2, 4, or 8 bytes wide. The shifting of the index register is done by the processor at the time the address is evaluated with no performance loss. This eliminates the need for a separate shift or multiply instruction.

The base, index, and displacement components may be used in any combination; any of these components may be null. A scale factor can be used only when an index is also used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly languages. Following are possible uses for some of the various combinations of address components.

DISPLACEMENT

The displacement alone indicates the offset of the operand. This combination is used to directly address a statically allocated scalar operand. An 8-bit, 16-bit, or 32-bit displacement can be used.

BASE

The offset of the operand is specified indirectly in one of the general registers, as for "based" variables.

BASE + DISPLACEMENT

A register and a displacement can be used together for two distinct purposes:

1. Index into static array when element size is not 2, 4, or 8 bytes. The displacement component encodes the offset of the beginning of the array. The register holds the results of a calculation to determine the offset of a specific element within the array.

2. Access item of a record. The displacement component locates an item within record. The base register selects one of several occurrences of record, thereby providing a compact encoding for this common function.

An important special case of this combination, is to access parameters in the procedure activation record in the stack. In this case, EBP is the best choice for the base register, because when EBP is used as a base register, the processor automatically uses the stack segment register (SS) to locate the operand, thereby providing a compact encoding for this common function.

(INDEX * SCALE) + DISPLACEMENT

This combination provides efficient indexing into a static array when the element size is 2, 4, or 8 bytes. The displacement addresses the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.

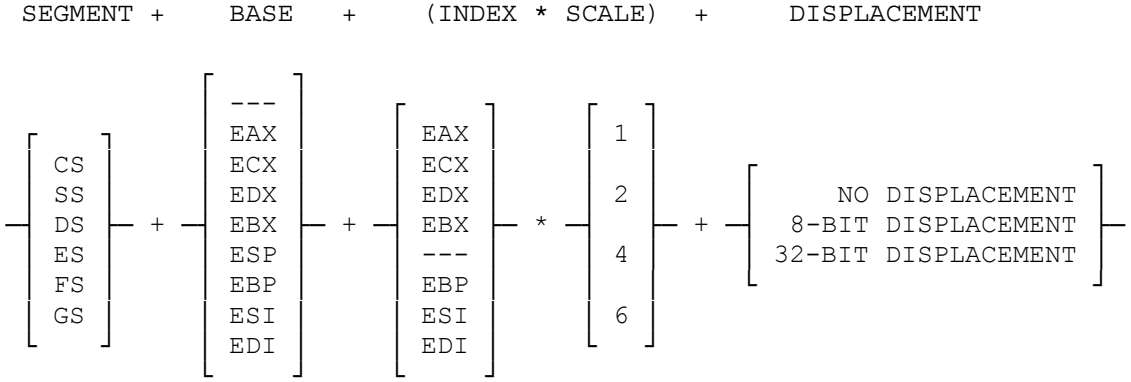
BASE + INDEX + DISPLACEMENT

Two registers used together support either a two-dimensional array (the displacement determining the beginning of the array) or one of several instances of an array of records (the displacement indicating an item in the record).

BASE + (INDEX * SCALE) + DISPLACEMENT

This combination provides efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes wide.

**Figure 2-10. Effective Address Computation**




## 2.6 Interrupts and Exceptions

The 80386 has two mechanisms for interrupting program execution:

1. Exceptions are synchronous events that are the responses of the CPU to certain conditions detected during the execution of an instruction.

2. Interrupts are asynchronous events typically triggered by external devices needing attention.

Interrupts and exceptions are alike in that both cause the processor to temporarily suspend its present program execution in order to execute a program of higher priority. The major distinction between these two kinds of interrupts is their origin. An exception is always reproducible by re-executing with the program and data that caused the exception, whereas an interrupt is generally independent of the currently executing program.

Application programmers are not normally concerned with servicing interrupts. More information on interrupts for systems programmers may be found in Chapter 9. Certain exceptions, however, are of interest to applications programmers, and many operating systems give applications programs the opportunity to service these exceptions. However, the operating system itself defines the interface between the applications programs and the exception mechanism of the 80386.

Table 2-2 highlights the exceptions that may be of interest to applications programmers.

- A divide error exception results when the instruction DIV or IDIV is executed with a zero denominator or when the quotient is too large for the destination operand. (Refer to Chapter 3 for a discussion of DIV and IDIV.)

- The debug exception may be reflected back to an applications program if it results from the trap flag (TF).

- A breakpoint exception results when the instruction INT 3 is executed. This instruction is used by some debuggers to stop program execution at specific points.

- An overflow exception results when the INTO instruction is executed and the OF (overflow) flag is set (after an arithmetic operation that set the OF flag). (Refer to Chapter 3 for a discussion of INTO).

- A bounds check exception results when the BOUND instruction is executed and the array index it checks falls outside the bounds of the array. (Refer to Chapter 3 for a discussion of the BOUND instruction.)

- Invalid opcodes may be used by some applications to extend the instruction set. In such a case, the invalid opcode exception presents an opportunity to emulate the opcode.

- The "coprocessor not available" exception occurs if the program contains instructions for a coprocessor, but no coprocessor is present in the system.

- A coprocessor error is generated when a coprocessor detects an illegal operation.

The instruction INT generates an interrupt whenever it is executed; the processor treats this interrupt as an exception. The effects of this interrupt (and the effects of all other exceptions) are determined by exception handler routines provided by the application program or as part of the systems software (provided by systems programmers). The INT instruction itself is discussed in Chapter 3. Refer to Chapter 9 for a more complete description of exceptions.

**Table 2-2. 80386 Reserved Exceptions and Interrupts**

| Vector Number | Description |
|---|---|
| 0 | Divide Error |
| 1 | Debug Exceptions |
| 2 | NMI Interrupt |
| 3 | Breakpoint |
| 4 | INTO Detected Overflow |
| 5 | BOUND Range Exceeded |
| 6 | Invalid Opcode |
| 7 | Coprocessor Not Available |
| 8 | Double Exception |
| 9 | Coprocessor Segment Overrun |
| 10 | Invalid Task State Segment |
| 11 | Segment Not Present |
| 12 | Stack Fault |
| 13 | General Protection |
| 14 | Page Fault |
| 15 | (reserved) |
| 16 | Coprocessor Error |
| 17-32 | (reserved) |

# Chapter 3 Applications Instruction Set

This chapter presents an overview of the instructions which programmers can use to write application software for the 80386 executing in protected virtual-address mode. The instructions are grouped by categories of related functions.

The instructions not discussed in this chapter are those that are normally used only by operating-system programmers. Part II describes the operation of these instructions.

The descriptions in this chapter assume that the 80386 is operating in protected mode with 32-bit addressing in effect; however, all instructions discussed are also available when 16-bit addressing is in effect in protected mode, real mode, or virtual 8086 mode. For any differences of operation that exist in the various modes, refer to Chapter 13, Chapter 14, or Chapter 15.

The instruction dictionary in Chapter 17 contains more detailed descriptions of all instructions, including encoding, operation, timing, effect on flags, and exceptions.

## 3.1 Data Movement Instructions

These instructions provide convenient methods for moving bytes, words, or doublewords of data between memory and the registers of the base architecture. They fall into the following classes:

1. General-purpose data movement instructions.
2. Stack manipulation instructions.
3. Type-conversion instructions.

### 3.1.1 General-Purpose Data Movement Instructions

MOV (Move) transfers a byte, word, or doubleword from the source operand to the destination operand. The MOV instruction is useful for transferring data along any of these paths. There are also variants of MOV that operate on segment registers. These are covered in a later section of this chapter.:

- To a register from memory
- To memory from a register
- Between general registers
- Immediate data to a register
- Immediate data to a memory

The MOV instruction cannot move from memory to memory or from segment register to segment register are not allowed. Memory-to-memory moves can be performed, however, by the string move instruction MOVS.

XCHG (Exchange) swaps the contents of two operands. This instruction takes the place of three MOV instructions. It does not require a temporary location to save the contents of one operand while load the other is being loaded. XCHG is especially useful for implementing semaphores or similar data structures for process synchronization.

The XCHG instruction can swap two byte operands, two word operands, or two doubleword operands. The operands for the XCHG instruction may be two register operands, or a register operand with a memory operand. When used with a memory operand, XCHG automatically activates the LOCK signal. (Refer to Chapter 11 for more information on the bus lock.)

### 3.1.2 Stack Manipulation Instructions

PUSH (Push) decrements the stack pointer (ESP), then transfers the source operand to the top of stack indicated by ESP (see Figure 3-1). PUSH is often used to place parameters on the stack before calling a procedure; it is also the basic means of storing temporary variables on the stack. The PUSH instruction operates on memory operands, immediate operands, and register operands (including segment registers).

PUSHA (Push All Registers) saves the contents of the eight general registers on the stack (see Figure 3-2). This instruction simplifies procedure calls by reducing the number of instructions required to retain the contents of the general registers for use in a procedure. The processor pushes the general registers on the stack in the following order: EAX, ECX, EDX, EBX, the initial value of ESP before EAX was pushed, EBP, ESI, and EDI. PUSHA is complemented by the POPA instruction.

POP (Pop) transfers the word or doubleword at the current top of stack (indicated by ESP) to the destination operand, and then increments ESP to point to the new top of stack. See Figure 3-3. POP moves information from the stack to a general register, or to memory
There are also a variant of POP that operates on segment registers. This is covered in a later section of this chapter..

POPA (Pop All Registers) restores the registers saved on the stack by PUSHA, except that it ignores the saved value of ESP. See Figure 3-4.

**Figure 3-1. PUSH**

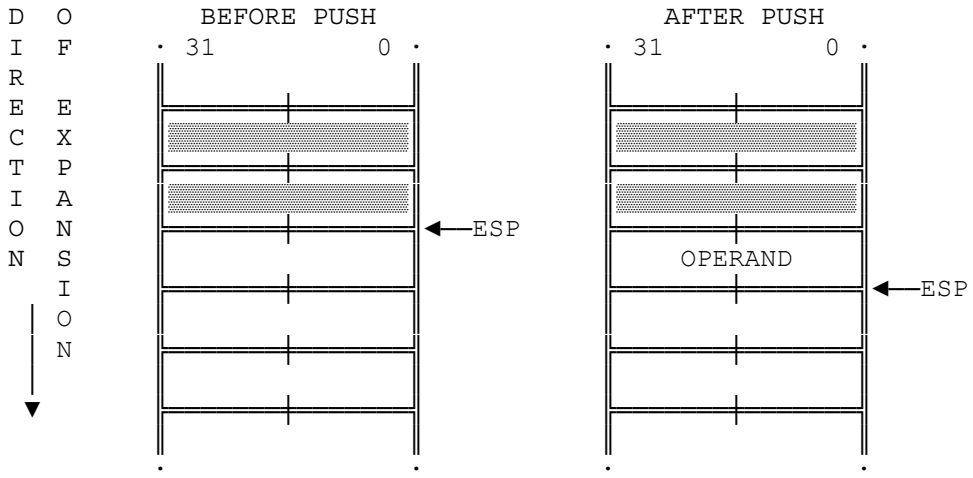
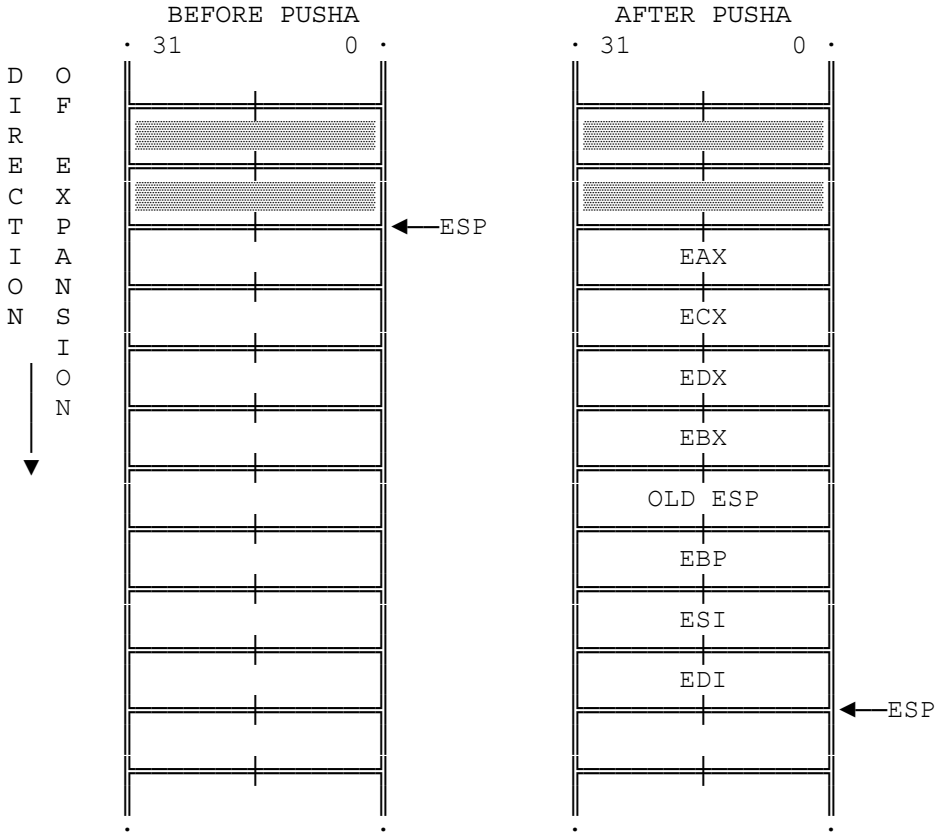



**Figure 3-2. PUSHA**



### 3.1.3 Type Conversion Instructions

The type conversion instructions convert bytes into words, words into doublewords, and doublewords into 64-bit items (quad-words). These instructions are especially useful for converting signed integers, because they automatically fill the extra bits of the larger item with the value of the sign bit of the smaller item. This kind of conversion, illustrated by Figure 3-5, is called sign extension.

There are two classes of type conversion instructions:

1. The forms CWD, CDQ, CBW, and CWDE which operate only on data in the EAX register.

2. The forms MOVSX and MOVZX, which permit one operand to be in any general register while permitting the other operand to be in memory or in a register.

CWD (Convert Word to Doubleword) and CDQ (Convert Doubleword to Quad-Word) double the size of the source operand. CWD extends the sign of the word in register AX throughout register DX. CDQ extends the sign of the doubleword in EAX throughout EDX. CWD can be used to produce a doubleword dividend from a word before a word division, and CDQ can be used to produce a quad-word dividend from a doubleword before doubleword division.

CBW (Convert Byte to Word) extends the sign of the byte in register AL throughout AX.

CWDE (Convert Word to Doubleword Extended) extends the sign of the word in register AX throughout EAX.

MOVSX (Move with Sign Extension) sign-extends an 8-bit value to a 16-bit value and a 8- or 16-bit value to 32-bit value.

MOVZX (Move with Zero Extension) extends an 8-bit value to a 16-bit value and an 8- or 16-bit value to 32-bit value by inserting high-order zeros.

**Figure 3-3. POP**

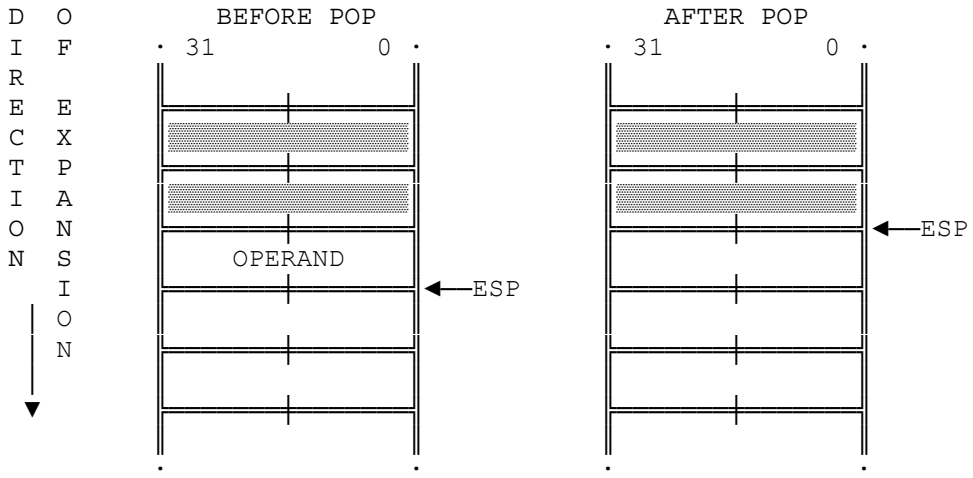



**Figure 3-4. POPA**

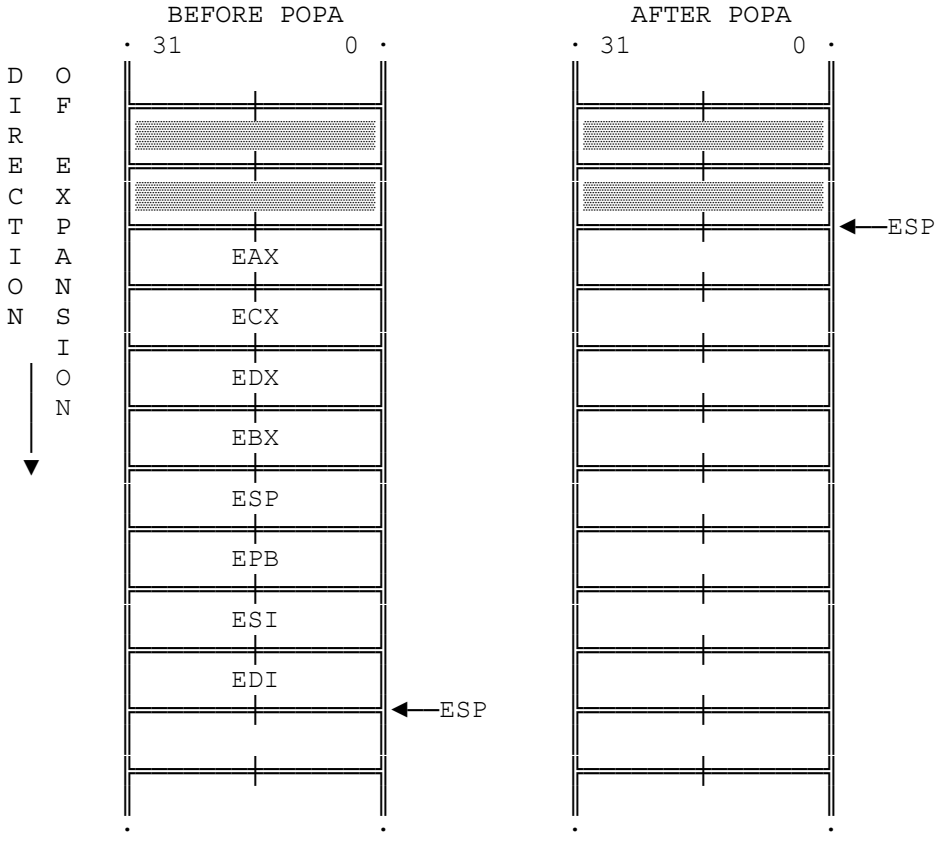
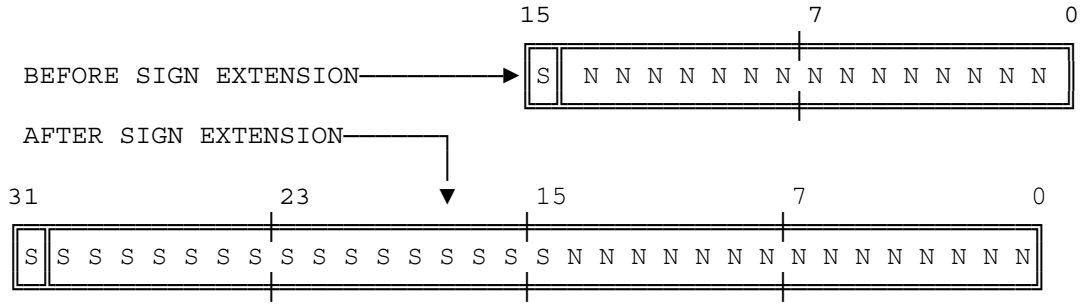


**Figure 3-5. Sign Extension**




## 3.2 Binary Arithmetic Instructions

The arithmetic instructions of the 80386 processor simplify the manipulation of numeric data that is encoded in binary. Operations include the standard add, subtract, multiply, and divide as well as increment, decrement, compare, and change sign. Both signed and unsigned binary integers are supported. The binary arithmetic instructions may also be used as one step in the process of performing arithmetic on decimal integers.

Many of the arithmetic instructions operate on both signed and unsigned integers. These instructions update the flags ZF, CF, SF, and OF in such a manner that subsequent instructions can interpret the results of the arithmetic as either signed or unsigned. CF contains information relevant to unsigned integers; SF and OF contain information relevant to signed integers. ZF is relevant to both signed and unsigned integers; ZF is set when all bits of the result are zero.

If the integer is unsigned, CF may be tested after one of these arithmetic operations to determine whether the operation required a carry or borrow of a one-bit in the high-order position of the destination operand. CF is set if a one-bit was carried out of the high-order position (addition instructions ADD, ADC, AAA, and DAA) or if a one-bit was carried (i.e. borrowed) into the high-order bit (subtraction instructions SUB, SBB, AAS, DAS, CMP, and NEG).

If the integer is signed, both SF and OF should be tested. SF always has the same value as the sign bit of the result. The most significant bit (MSB) of a signed integer is the bit next to the sign—bit 6 of a byte, bit 14 of a word, or bit 30 of a doubleword. OF is set in either of these cases:

- A one-bit was carried out of the MSB into the sign bit but no one bit was carried out of the sign bit (addition instructions ADD, ADC, INC, AAA, and DAA). In other words, the result was greater than the greatest positive number that could be contained in the destination operand.

- A one-bit was carried from the sign bit into the MSB but no one bit was carried into the sign bit (subtraction instructions SUB, SBB, DEC, AAS, DAS, CMP, and NEG). In other words, the result was smaller that the smallest negative number that could be contained in the destination operand.

These status flags are tested by executing one of the two families of conditional instructions: Jcc (jump on condition cc) or SETcc (byte set on condition).

### 3.2.1 Addition and Subtraction Instructions

ADD (Add Integers) replaces the destination operand with the sum of the source and destination operands. Sets CF if overflow.

ADC (Add Integers with Carry) sums the operands, adds one if CF is set, and replaces the destination operand with the result. If CF is cleared, ADC performs the same operation as the ADD instruction. An ADD followed by multiple ADC instructions can be used to add numbers longer than 32 bits.

INC (Increment) adds one to the destination operand. INC does not affect CF. Use ADD with an immediate value of 1 if an increment that updates carry (CF) is needed.

SUB (Subtract Integers) subtracts the source operand from the destination operand and replaces the destination operand with the result. If a borrow is required, the CF is set. The operands may be signed or unsigned bytes, words, or doublewords.

SBB (Subtract Integers with Borrow) subtracts the source operand from the destination operand, subtracts 1 if CF is set, and returns the result to the destination operand. If CF is cleared, SBB performs the same operation as SUB. SUB followed by multiple SBB instructions may be used to subtract numbers longer than 32 bits. If CF is cleared, SBB performs the same operation as SUB.

DEC (Decrement) subtracts 1 from the destination operand. DEC does not update CF. Use SUB with an immediate value of 1 to perform a decrement that affects carry.

### 3.2.2 Comparison and Sign Change Instruction

CMP (Compare) subtracts the source operand from the destination operand. It updates OF, SF, ZF, AF, PF, and CF but does not alter the source and destination operands. A subsequent Jcc or SETcc instruction can test the appropriate flags.

NEG (Negate) subtracts a signed integer operand from zero. The effect of NEG is to reverse the sign of the operand from positive to negative or from negative to positive.

### 3.2.3 Multiplication Instructions

The 80386 has separate multiply instructions for unsigned and signed operands. MUL operates on unsigned numbers, while IMUL operates on signed integers as well as unsigned.

MUL (Unsigned Integer Multiply) performs an unsigned multiplication of the source operand and the accumulator. If the source is a byte, the processor multiplies it by the contents of AL and returns the double-length result to AH and AL. If the source operand is a word, the processor multiplies it by the contents of AX and returns the double-length result to DX and AX. If the source operand is a doubleword, the processor multiplies it by the contents of EAX and returns the 64-bit result in EDX and EAX. MUL sets CF and OF when the upper half of the result is nonzero; otherwise, they are cleared.

IMUL (Signed Integer Multiply) performs a signed multiplication operation. IMUL has three variations:

1. A one-operand form. The operand may be a byte, word, or doubleword located in memory or in a general register. This instruction uses EAX and EDX as implicit operands in the same way as the MUL instruction.

2. A two-operand form. One of the source operands may be in any general register while the other may be either in memory or in a general register. The product replaces the general-register operand.

3. A three-operand form; two are source and one is the destination operand. One of the source operands is an immediate value stored in the instruction; the second may be in memory or in any general register. The product may be stored in any general register. The immediate operand is treated as signed. If the immediate operand is a byte, the processor automatically sign-extends it to the size of the second operand before performing the multiplication.

The three forms are similar in most respects:

- The length of the product is calculated to twice the length of the operands.

- The CF and OF flags are set when significant bits are carried into the high-order half of the result. CF and OF are cleared when the high-order half of the result is the sign-extension of the low-order half.

However, forms 2 and 3 differ in that the product is truncated to the length of the operands before it is stored in the destination register. Because of this truncation, OF should be tested to ensure that no significant bits are lost. (For ways to test OF, refer to the INTO and PUSHF instructions.)

Forms 2 and 3 of IMUL may also be used with unsigned operands because, whether the operands are signed or unsigned, the low-order half of the product is the same.

### 3.2.4 Division Instructions

The 80386 has separate division instructions for unsigned and signed operands. DIV operates on unsigned numbers, while IDIV operates on signed integers as well as unsigned. In either case, an exception (interrupt zero) occurs if the divisor is zero or if the quotient is too large for AL, AX, or EAX.

DIV (Unsigned Integer Divide) performs an unsigned division of the accumulator by the source operand. The dividend (the accumulator) is twice the size of the divisor (the source operand); the quotient and remainder have the same size as the divisor, as the following table shows.

| Size of Source Operand (divisor) | Dividend | Quotient | Remainder |
|---|---|---|---|
| Byte | AX | AL | AH |
| Word | DX:AX | AX | DX |
| Doubleword | EDX:EAX | EAX | EDX |

Non-integral quotients are truncated to integers toward 0. The remainder is always less than the divisor. For unsigned byte division, the largest quotient is 255. For unsigned word division, the largest quotient is 65,535. For unsigned doubleword division the largest quotient is $2^{32}$-1.

IDIV (Signed Integer Divide) performs a signed division of the accumulator by the source operand. IDIV uses the same registers as the DIV instruction.

For signed byte division, the maximum positive quotient is +127, and the minimum negative quotient is -128. For signed word division, the maximum positive quotient is +32,767, and the minimum negative quotient is -32,768. For signed doubleword division the maximum positive quotient is $2^{31}$-1, the minimum negative quotient is $-2^{31}$. Non-integral results are truncated towards 0. The remainder always has the same sign as the dividend and is less than the divisor in magnitude.

## 3.3 Decimal Arithmetic Instructions

Decimal arithmetic is performed by combining the binary arithmetic instructions (already discussed in the prior section) with the decimal arithmetic instructions. The decimal arithmetic instructions are used in one of the following ways:

- To adjust the results of a previous binary arithmetic operation to produce a valid packed or unpacked decimal result.
- To adjust the inputs to a subsequent binary arithmetic operation so that the operation will produce a valid packed or unpacked decimal result.

These instructions operate only on the AL or AH registers. Most utilize the AF flag.

### 3.3.1 Packed BCD Adjustment Instructions

DAA (Decimal Adjust after Addition) adjusts the result of adding two valid packed decimal operands in AL. DAA must always follow the addition of two pairs of packed decimal numbers (one digit in each half-byte) to obtain a pair of valid packed decimal digits as results. The carry flag is set if carry was needed.

DAS (Decimal Adjust after Subtraction) adjusts the result of subtracting two valid packed decimal operands in AL. DAS must always follow the subtraction of one pair of packed decimal numbers (one digit in each half-byte) from another to obtain a pair of valid packed decimal digits as results. The carry flag is set if a borrow was needed.

### 3.3.2 Unpacked BCD Adjustment Instructions

AAA (ASCII Adjust after Addition) changes the contents of register AL to a valid unpacked decimal number, and zeros the top 4 bits. AAA must always follow the addition of two unpacked decimal operands in AL. The carry flag is set and AH is incremented if a carry is necessary.

AAS (ASCII Adjust after Subtraction) changes the contents of register AL to a valid unpacked decimal number, and zeros the top 4 bits. AAS must always follow the subtraction of one unpacked decimal operand from another in AL. The carry flag is set and AH decremented if a borrow is necessary.

AAM (ASCII Adjust after Multiplication) corrects the result of a multiplication of two valid unpacked decimal numbers. AAM must always follow the multiplication of two decimal numbers to produce a valid decimal result. The high order digit is left in AH, the low order digit in AL.

AAD (ASCII Adjust before Division) modifies the numerator in AH and AL to prepare for the division of two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. AH should contain the high-order digit and AL the low-order digit. This instruction adjusts the value and places the result in AL. AH will contain zero.

## 3.4 Logical Instructions

The group of logical instructions includes:

- The Boolean operation instructions
- Bit test and modify instructions
- Bit scan instructions
- Rotate and shift instructions
- Byte set on condition

### 3.4.1 Boolean Operation Instructions

The logical operations are AND, OR, XOR, and NOT.

NOT (Not) inverts the bits in the specified operand to form a one's complement of the operand. The NOT instruction is a unary operation that uses a single operand in a register or memory. NOT has no effect on the flags.

The AND, OR, and XOR instructions perform the standard logical operations "and", "(inclusive) or", and "exclusive or". These instructions can use the following combinations of operands:

- Two register operands
- A general register operand with a memory operand
- An immediate operand with either a general register operand or a memory operand.

AND, OR, and XOR clear OF and CF, leave AF undefined, and update SF, ZF, and PF.

### 3.4.2 Bit Test and Modify Instructions

This group of instructions operates on a single bit which can be in memory or in a general register. The location of the bit is specified as an offset from the low-order end of the operand. The value of the offset either may be given by an immediate byte in the instruction or may be contained in a general register.

These instructions first assign the value of the selected bit to CF, the carry flag. Then a new value is assigned to the selected bit, as determined by the operation. OF, SF, ZF, AF, PF are left in an undefined state. Table 3-1 defines these instructions.

**Table 3-1. Bit Test and Modify Instructions**

| Instruction | Effect on CF | Effect on Selected Bit |
|---|---|---|
| Bit (Bit Test) | CF ← BIT | (none) |
| BTS (Bit Test and Set) | CF ← BIT | BIT ← 1 |
| BTR (Bit Test and Reset) | CF ← BIT | BIT ← 0 |
| BTC (Bit Test and Complement) | CF ← BIT | BIT ← NOT(BIT) |

### 3.4.3 Bit Scan Instructions

These instructions scan a word or doubleword for a one-bit and store the index of the first set bit into a register. The bit string being scanned may be either in a register or in memory. The ZF flag is set if the entire word is zero (no set bits are found); ZF is cleared if a one-bit is found. If no set bit is found, the value of the destination register is undefined.

BSF (Bit Scan Forward) scans from low-order to high-order (starting from bit index zero).

BSR (Bit Scan Reverse) scans from high-order to low-order (starting from bit index 15 of a word or index 31 of a doubleword).

### 3.4.4 Shift and Rotate Instructions

The shift and rotate instructions reposition the bits within the specified operand.

These instructions fall into the following classes:

- Shift instructions
- Double shift instructions
- Rotate instructions

#### 3.4.4.1 Shift Instructions

The bits in bytes, words, and doublewords may be shifted arithmetically or logically. Depending on the value of a specified count, bits can be shifted up to 31 places.

A shift instruction can specify the count in one of three ways. One form of shift instruction implicitly specifies the count as a single shift. The second form specifies the count as an immediate value. The third form specifies the count as the value contained in CL. This last form allows the shift count to be a variable that the program supplies during execution. Only the low order 5 bits of CL are used.

CF always contains the value of the last bit shifted out of the destination operand. In a single-bit shift, OF is set if the value of the high-order (sign) bit was changed by the operation. Otherwise, OF is cleared. Following a multibit shift, however, the content of OF is always undefined.

The shift instructions provide a convenient way to accomplish division or multiplication by binary power. Note however that division of signed numbers by shifting right is not the same kind of division performed by the IDIV instruction.

SAL (Shift Arithmetic Left) shifts the destination byte, word, or doubleword operand left by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL). The processor shifts zeros in from the right (low-order) side of the operand as bits exit from the left (high-order) side. See Figure 3-6.

SHL (Shift Logical Left) is a synonym for SAL (refer to SAL).

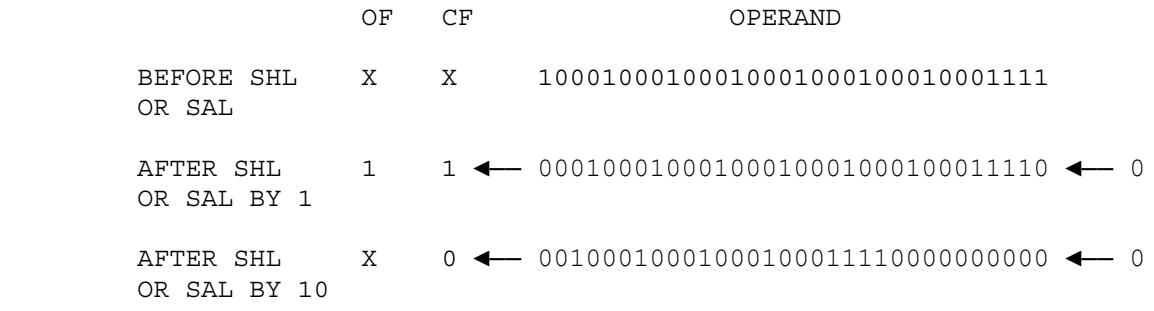
SHR (Shift Logical Right) shifts the destination byte, word, or doubleword operand right by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL). The processor shifts zeros in from the left side of the operand as bits exit from the right side. See Figure 3-7.

SAR (Shift Arithmetic Right) shifts the destination byte, word, or doubleword operand to the right by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL). The processor preserves the sign of the operand by shifting in zeros on the left (high-order) side if the value is positive or by shifting by ones if the value is negative. See Figure 3-8.

Even though this instruction can be used to divide integers by a power of two, the type of division is not the same as that produced by the IDIV instruction. The quotient of IDIV is rounded toward zero, whereas the "quotient" of SAR is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when IDIV is used to divide -9 by 4, the result is -2 with a remainder of -1. If SAR is used to shift -9 right by two bits, the result is -3. The "remainder" of this kind of division is +3; however, the SAR instruction stores only the high-order bit of the remainder (in CF).

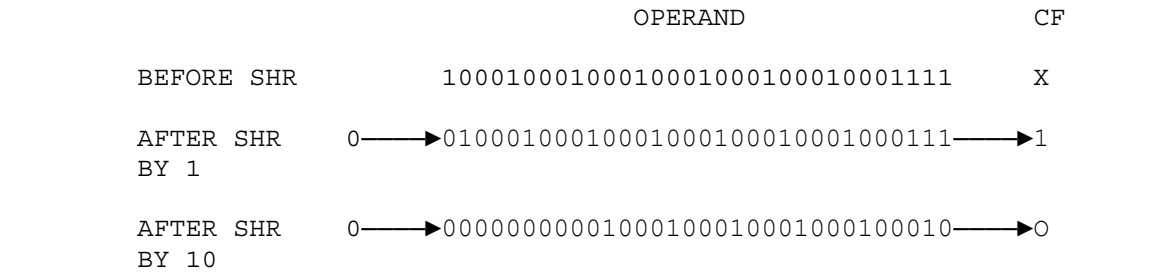
The code sequence in Figure 3-9 produces the same result as IDIV for any M = $2^N$, where 0 < N < 32. This sequence takes about 12 to 18 clocks, depending on whether the jump is taken; if ECX contains M, the corresponding IDIV ECX instruction will take about 43 clocks.

**Figure 3-6. SAL and SHL**



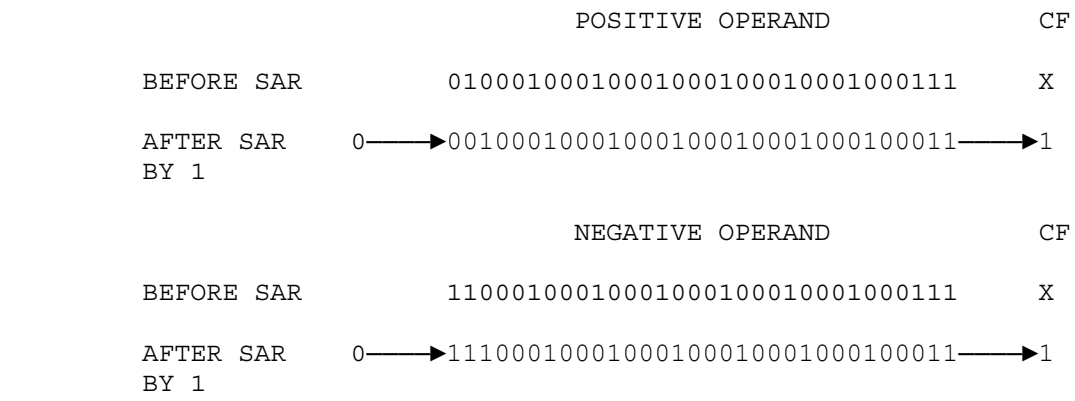

SHL (WHICH HAS THE SYNONYM SAL) SHIFTS THE BITS IN THE REGISTER OR MEMORY OPERAND TO THE LEFT BY THE SPECIFIED NUMBER OF BIT POSITIONS. CF RECEIVES THE LAST BIT SHIFTED OUT OF THE LEFT OF THE OPERAND. SHL SHIFTS IN ZEROS TO FILL THE VACATED BIT LOCATIONS. THESE INSTRUCTIONS OPERATE ON BYTE, WORD, AND DOUBLEWORD OPERANDS.

**Figure 3-7. SHR**




SHR SHIFTS THE BITS OF THE REGISTER OR MEMORY OPERAND TO THE RIGHT BY THE SPECIFIED NUMBER OF BIT POSITIONS. CF RECEIVES THE LAST BIT SHIFTED OUT OF THE RIGHT OF THE OPERAND. SHR SHIFTS IN ZEROS TO FILL THE VACATED BIT LOCATIONS.

**Figure 3-8. SAR**




SAR PRESERVES THE SIGN OF THE REGISTER OR MEMORY OPERAND AS IT SHIFTS THE OPERAND TO THE RIGHT BY THE SPECIFIED NUMBER OF BIT POSITIONS. CF RECIEVES THE LAST BIT SHIFTED OUT OF THE RIGHT OF THE OPERAND.

**Figure 3-9. Using SAR to Simulate IDIV**

```
    ; assuming N is in ECX, and the dividend is in EAX
    ;                                                CLOCKS
    CMP     EAX, 0      ; to set sign flag           2
    JGE     NoAdjust    ; jump if sign is zero       3 or 9
    ADD     EAX, ECX    ;                            2
    DEC     EAX         ; EAX := EAX + (N-1)         2
NoAdjust:
    SAR     EAX, CL     ;                            3
    ;                       TOTAL CLOCKS            12 or 18]
```

### 3.4.4.2 Double-Shift Instructions

These instructions provide the basic operations needed to implement operations on long unaligned bit strings. The double shifts operate either on word or doubleword operands, as follows:

1. Taking two word operands as input and producing a one-word output.

2. Taking two doubleword operands as input and producing a doubleword output.

Of the two input operands, one may either be in a general register or in memory, while the other may only be in a general register. The results replace the memory or register operand. The number of bits to be shifted is specified either in the CL register or in an immediate byte of the instruction.

Bits are shifted from the register operand into the memory or register operand. CF is set to the value of the last bit shifted out of the destination operand. SF, ZF, and PF are set according to the value of the result. OF and AF are left undefined.

SHLD (Shift Left Double) shifts bits of the R/M field to the left, while shifting high-order bits from the Reg field into the R/M field on the right (see Figure 3-10). The result is stored back into the R/M operand. The Reg field is not modified.

SHRD (Shift Right Double) shifts bits of the R/M field to the right, while shifting low-order bits from the Reg field into the R/M field on the left (see Figure 3-11). The result is stored back into the R/M operand. The Reg field is not modified.

### 3.4.4.3 Rotate Instructions

Rotate instructions allow bits in bytes, words, and doublewords to be rotated. Bits rotated out of an operand are not lost as in a shift, but are "circled" back into the other "end" of the operand.

Rotates affect only the carry and overflow flags. CF may act as an extension of the operand in two of the rotate instructions, allowing a bit to be isolated and then tested by a conditional jump instruction (JC or JNC). CF always contains the value of the last bit rotated out, even if the instruction does not use this bit as an extension of the rotated operand.

In single-bit rotates, OF is set if the operation changes the high-order (sign) bit of the destination operand. If the sign bit retains its original value, OF is cleared. On multibit rotates, the value of OF is always undefined.

ROL (Rotate Left) rotates the byte, word, or doubleword destination operand left by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL). For each rotation specified, the high-order bit that exits from the left of the operand returns at the right to become the new low-order bit of the operand. See Figure 3-12.

ROR (Rotate Right) rotates the byte, word, or doubleword destination operand right by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL). For each rotation specified, the low-order bit that exits from the right of the operand returns at the left to become the new high-order bit of the operand. See Figure 3-13.

RCL (Rotate Through Carry Left) rotates bits in the byte, word, or doubleword destination operand left by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL).

This instruction differs from ROL in that it treats CF as a high-order one-bit extension of the destination operand. Each high-order bit that exits from the left side of the operand moves to CF before it returns to the operand as the low-order bit on the next rotation cycle. See Figure 3-14.

RCR (Rotate Through Carry Right) rotates bits in the byte, word, or doubleword destination operand right by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL).

This instruction differs from ROR in that it treats CF as a low-order one-bit extension of the destination operand. Each low-order bit that exits from the right side of the operand moves to CF before it returns to the operand as the high-order bit on the next rotation cycle. See Figure 3-15.

**Figure 3-10. Shift Left Double**

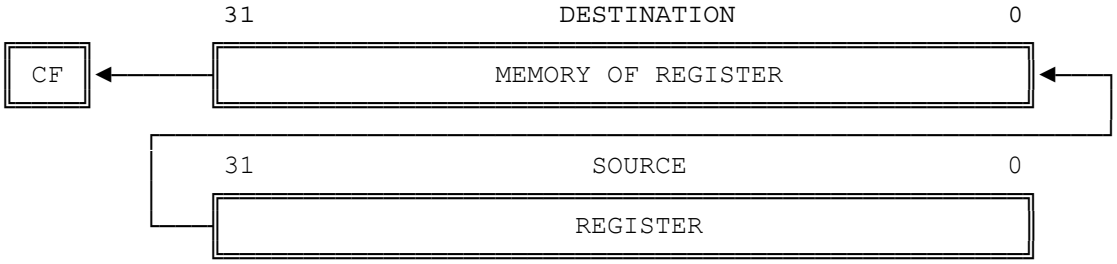



**Figure 3-11. Shift Right Double**

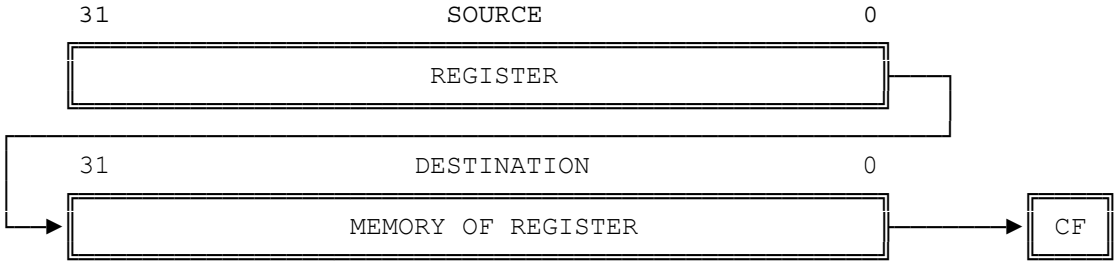



**Figure 3-12. ROL**

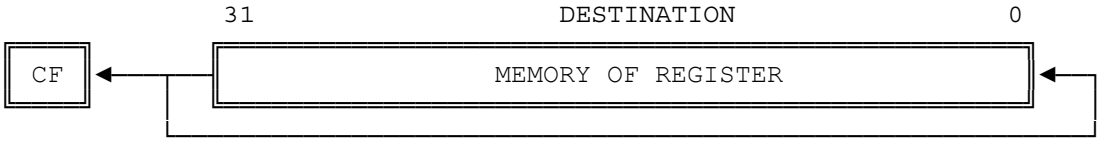



**Figure 3-13. ROR**

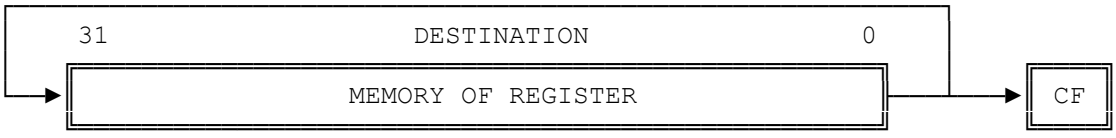



**Figure 3-14. RCL**

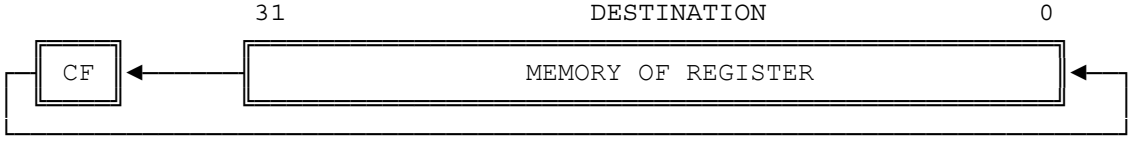


**Figure 3-15. RCR**


### 3.4.4.4 Fast "BIT BLT" Using Double Shift Instructions

One purpose of the double shifts is to implement a bit string move, with arbitrary misalignment of the bit strings. This is called a "bit blt" (BIT BLock Transfer.) A simple example is to move a bit string from an arbitrary offset into a doubleword-aligned byte string. A left-to-right string is moved 32 bits at a time if a double shift is used inside the move loop.

```
     MOV   ESI,ScrAddr
     MOV   EDI,DestAddr
     MOV   EBX,WordCnt
     MOV   CL,RelOffset      ; relative offset Dest-Src
     MOV   EDX,[ESI]         ; load first word of source
     ADD   ESI,4             ; bump source address
BltLoop:
     LODS                    ; new low order part
     SHLD  EDX,EAX,CL        ; EDX overwritten with aligned stuff
     XCHG  EDX,EAS           ; Swap high/low order parts
     STOS                    ; Write out next aligned chunk
     DEC   EBX
     JA    BltLoop
```

This loop is simple yet allows the data to be moved in 32-bit pieces for the highest possible performance. Without a double shift, the best that can be achieved is 16 bits per loop iteration by using a 32-bit shift and replacing the XCHG with a ROR by 16 to swap high and low order parts of registers. A more general loop than shown above would require some extra masking on the first doubleword moved (before the main loop), and on the last doubleword moved (after the main loop), but would have the same basic 32-bits per loop iteration as the code above.

### 3.4.4.5 Fast Bit-String Insert and Extract

The double shift instructions also enable:

- Fast insertion of a bit string from a register into an arbitrary bit location in a larger bit string in memory without disturbing the bits on either side of the inserted bits.
- Fast extraction of a bits string into a register from an arbitrary bit location in a larger bit string in memory without disturbing the bits on either side of the extracted bits.

The following coded examples illustrate bit insertion and extraction under variousconditions:

1. Bit String Insert into Memory (when bit string is 1-25 bits long, i.e., spans four bytes or less):

```
; Insert a right-justified bit string from register into
; memory bit string.
;
; Assumptions:
; 1) The base of the string array is dword aligned, and
; 2) the length of the bit string is an immediate value
;    but the bit offset is held in a register.
;
; Register ESI holds the right-justified bit string
; to be inserted.
; Register EDI holds the bit offset of the start of the
; substring.
; Registers EAX and ECX are also used by this
; "insert" operation.
;
MOV   ECX,EDI      ; preserve original offset for later use
SHR   EDI,3        ; signed divide offset by 8 (byte address)
AND   CL,7H        ; isolate low three bits of offset in CL
MOV   EAX,[EDI]strg_base      ; move string dword into EAX
ROR   EAX,CL       ; right justify old bit field
SHRD  EAX,ESI,length          ; bring in new bits
ROL   EAX,length   ; right justify new bit field
ROL   EAX,CL       ; bring to final position
MOV   [EDI]strg_base,EAX      ; replace dword in memory
```

2. Bit String Insert into Memory (when bit string is 1-31 bits long, i.e. spans five bytes or less):

```
; Insert a right-justified bit string from register into
; memory bit string.
;
; Assumptions:
; 1) The base of the string array is dword aligned, and
; 2) the length of the bit string is an immediate value
;    but the bit offset is held in a register.
;
; Register ESI holds the right-justified bit string
; to be inserted.
; Register EDI holds the bit offset of the start of the
; substring.
; Registers EAX, EBX, ECX, and EDI are also used by
; this "insert" operation.
;
MOV   ECX,EDI     ; temp storage for offset
SHR   EDI,5       ; signed divide offset by 32 (dword address)
SHL   EDI,2       ; multiply by 4 (in byte address format)
AND   CL,1FH      ; isolate low five bits of offset in CL
MOV   EAX,[EDI]strg_base      ; move low string dword into EAX
MOV   EDX,[EDI]strg_base+4    ; other string dword into EDX
MOV   EBX,EAX     ; temp storage for part of string     ┐ rotate
SHRD  EAX,EDX,CL  ; double shift by offset within dword ├ EDX:EAX
```

```
SHRD  EAX,EBX,CL  ; double shift by offset within dword ┘ right
SHRD  EAX,ESI,length          ; bring in new bits
ROL   EAX,length  ; right justify new bit field
MOV   EBX,EAX     ; temp storage for part of string         ┐ rotate
SHLD  EAX,EDX,CL  ; double shift back by offset within word ├ EDX:EAX
SHLD  EDX,EBX,CL  ; double shift back by offset within word ┘ left
MOV   [EDI]strg_base,EAX      ; replace dword in memory
MOV   [EDI]strg_base+4,EDX    ; replace dword in memory
```

3. Bit String Insert into Memory (when bit string is exactly 32 bits long, i.e., spans five or four types of memory):

```
; Insert right-justified bit string from register into
; memory bit string.
;
; Assumptions:
; 1) The base of the string array is dword aligned, and
; 2) the length of the bit string is 32
;    but the bit offset is held in a register.
;
; Register ESI holds the 32-bit string to be inserted.
; Register EDI holds the bit offset of the start of the
; substring.
; Registers EAX, EBX, ECX, and EDI are also used by
; this "insert" operation.
;
MOV   EDX,EDI     ; preserve original offset for later use
SHR   EDI,5       ; signed divide offset by 32 (dword address)
SHL   EDI,2       ; multiply by 4 (in byte address format)
AND   CL,1FH      ; isolate low five bits of offset in CL
MOV   EAX,[EDI]strg_base      ; move low string dword into EAX
MOV   EDX,[EDI]strg_base+4    ; other string dword into EDX
MOV   EBX,EAX     ; temp storage for part of string     ┐ rotate
SHRD  EAX,EDX     ; double shift by offset within dword ├ EDX:EAX
SHRD  EDX,EBX     ; double shift by offset within dword ┘ right
MOV   EAX,ESI     ; move 32-bit bit field into position
MOV   EBX,EAX     ; temp storage for part of string         ┐ rotate
SHLD  EAX,EDX     ; double shift back by offset within word ├ EDX:EAX
SHLD  EDX,EBX     ; double shift back by offset within word ┘ left
MOV   [EDI]strg_base,EAX      ; replace dword in memory
MOV   [EDI]strg_base,+4,EDX   ; replace dword in memory
```

4. Bit String Extract from Memory (when bit string is 1-25 bits long, i.e., spans four bytes or less):

```
; Extract a right-justified bit string from memory bit
; string into register
;
; Assumptions:
; 1) The base of the string array is dword aligned, and
; 2) the length of the bit string is an immediate value
;    but the bit offset is held in a register.
;
; Register EAX holds the right-justified, zero-padded
; bit string that was extracted.
; Register EDI holds the bit offset of the start of the
; substring.
```

```
; Registers EDI, and ECX are also used by this "extract."
;
MOV  ECX,EDI      ; temp storage for offset
SHR  EDI,3        ; signed divide offset by 8 (byte address)
AND  CL,7H        ; isolate low three bits of offset
MOV  EAX,[EDI]strg_base       ; move string dword into EAX
SHR  EAX,CL       ; shift by offset within dword
AND  EAX,mask     ; extracted bit field in EAX
```

5. Bit String Extract from Memory (when bit string is 1-32 bits long, i.e., spans five bytes or less):

```
; Extract a right-justified bit string from memory bit
; string into register.
;
; Assumptions:
; 1) The base of the string array is dword aligned, and
; 2) the length of the bit string is an immediate
;    value but the bit offset is held in a register.
;
; Register EAX holds the right-justified, zero-padded
; bit string that was extracted.
; Register EDI holds the bit offset of the start of the
; substring.
; Registers EAX, EBX, and ECX are also used by this "extract."
MOV   ECX,EDI     ; temp storage for offset
SHR   EDI,5       ; signed divide offset by 32 (dword address)
SHL   EDI,2       ; multiply by 4 (in byte address format)
AND   CL,1FH      ; isolate low five bits of offset in CL
MOV   EAX,[EDI]strg_base      ; move low string dword into EAX
MOV   EDX,[EDI]strg_base+4    ; other string dword into EDX
SHRD  EAX,EDX,CL  ; double shift right by offset within dword
AND   EAX,mask    ; extracted bit field in EAX
```

### 3.4.5 Byte-Set-On-Condition Instructions

This group of instructions sets a byte to zero or one depending on any of the 16 conditions defined by the status flags. The byte may be in memory or may be a one-byte general register. These instructions are especially useful for implementing Boolean expressions in high-level languages such as Pascal.

SETcc (Set Byte on Condition cc) set a byte to one if condition cc is true; sets the byte to zero otherwise. Refer to Appendix D for a definition of the possible conditions.

### 3.4.6 Test Instruction

TEST (Test) performs the logical "and" of the two operands, clears OF and CF, leaves AF undefined, and updates SF, ZF, and PF. The flags can be tested by conditional control transfer instructions or by the byte-set-on-condition instructions. The operands may be doublewords, words, or bytes.

The difference between TEST and AND is that TEST does not alter the destination operand. TEST differs from BT in that TEST is useful for testing the value of multiple bits in one operations, whereas BT tests a single bit.

## 3.5 Control Transfer Instructions

The 80386 provides both conditional and unconditional control transfer instructions to direct the flow of execution. Conditional control transfers depend on the results of operations that affect the flag register. Unconditional control transfers are always executed.

### 3.5.1 Unconditional Transfer Instructions

JMP, CALL, RET, INT and IRET instructions transfer control from one code segment location to another. These locations can be within the same code segment (near control transfers) or in different code segments (far control transfers). The variants of these instructions that transfer control to other segments are discussed in a later section of this chapter. If the model of memory organization used in a particular 80386 application does not make segments visible to applications programmers, intersegment control transfers will not be used.

#### 3.5.1.1 Jump Instruction

JMP (Jump) unconditionally transfers control to the target location. JMP is a one-way transfer of execution; it does not save a return address on the stack.

The JMP instruction always performs the same basic function of transferring control from the current location to a new location. Its implementation varies depending on whether the address is specified directly within the instruction or indirectly through a register or memory.

A direct JMP instruction includes the destination address as part of the instruction. An indirect JMP instruction obtains the destination address indirectly through a register or a pointer variable.

Direct near JMP. A direct JMP uses a relative displacement value contained in the instruction. The displacement is signed and the size of the displacement may be a byte, word, or doubleword. The processor forms an effective address by adding this relative displacement to the address contained in EIP. When the additions have been performed, EIP refers to the next instruction to be executed.

Indirect near JMP. Indirect JMP instructions specify an absolute address in one of several ways:

1. The program can JMP to a location specified by a general register (any of EAX, EDX, ECX, EBX, EBP, ESI, or EDI). The processor moves this 32-bit value into EIP and resumes execution.

2. The processor can obtain the destination address from a memory operand specified in the instruction.

3. A register can modify the address of the memory pointer to select a destination address.

#### 3.5.1.2 Call Instruction

CALL (Call Procedure) activates an out-of-line procedure, saving on the stack the address of the instruction following the CALL for later use by a RET (Return) instruction. CALL places the current value of EIP on the stack. The RET instruction in the called procedure uses this address to transfer control back to the calling program.

CALL instructions, like JMP instructions have relative, direct, and indirect versions.

Indirect CALL instructions specify an absolute address in one of these ways:

1. The program can CALL a location specified by a general register (any of EAX, EDX, ECX, EBX, EBP, ESI, or EDI). The processor moves this 32-bit value into EIP.

2. The processor can obtain the destination address from a memory operand specified in the instruction.

#### 3.5.1.3 Return and Return-From-Interrupt Instruction

RET (Return From Procedure) terminates the execution of a procedure and transfers control through a back-link on the stack to the program that originally invoked the procedure. RET restores the value of EIP that was saved on the stack by the previous CALL instruction.

RET instructions may optionally specify an immediate operand. By adding this constant to the new top-of-stack pointer, RET effectively removes any arguments that the calling program pushed on the stack before the execution of the CALL instruction.

IRET (Return From Interrupt) returns control to an interrupted procedure. IRET differs from RET in that it also pops the flags from the stack into the flags register. The flags are stored on the stack by the interrupt mechanism.

### 3.5.2 Conditional Transfer Instructions

The conditional transfer instructions are jumps that may or may not transfer control, depending on the state of the CPU flags when the instruction executes.

### 3.5.2.1 Conditional Jump Instructions

Table 3-2 shows the conditional transfer mnemonics and their interpretations. The conditional jumps that are listed as pairs are actually the same instruction. The assembler provides the alternate mnemonics for greater clarity within a program listing.

Conditional jump instructions contain a displacement which is added to the EIP register if the condition is true. The displacement may be a byte, a word, or a doubleword. The displacement is signed; therefore, it can be used to jump forward or backward.

**Table 3-2. Interpretation of Conditional Transfers**

Unsigned Conditional Transfers

| Mnemonic | Condition Tested | "Jump If..." |
|---|---|---|
| JA/JNBE | (CF or ZF) = 0 | above/not below nor equal |
| JAE/JNB | CF = 0 | above or equal/not below |
| JB/JNAE | CF = 1 | below/not above nor equal |
| JBE/JNA | (CF or ZF) = 1 | below or equal/not above |
| JC | CF = 1 | carry |
| JE/JZ | ZF = 1 | equal/zero |
| JNC | CF = 0 | not carry |
| JNE/JNZ | ZF = 0 | not equal/not zero |
| JNP/JPO | PF = 0 | not parity/parity odd |
| JP/JPE | PF = 1 | parity/parity even |

Signed Conditional Transfers

| Mnemonic | Condition Tested | "Jump If..." |
|---|---|---|
| JG/JNLE | ((SF xor OF) or ZF) = 0 | greater/not less nor equal |
| JGE/JNL | (SF xor OF) = 0 | greater or equal/not less |
| JL/JNGE | (SF xor OF) = 1 | less/not greater nor equal |
| JLE/JNG | ((SF xor OF) or ZF) = 1 | less or equal/not greater |
| JNO | OF = 0 | not overflow |
| JNS | SF = 0 | not sign (positive, including 0) |
| JO | OF = 1 | overflow |
| JS | SF = 1 | sign (negative) |

### 3.5.2.2 Loop Instructions

The loop instructions are conditional jumps that use a value placed in ECX to specify the number of repetitions of a software loop. All loop instructions automatically decrement ECX and terminate the loop when ECX=0. Four of the five loop instructions specify a condition involving ZF that terminates the loop before ECX reaches zero.

LOOP (Loop While ECX Not Zero) is a conditional transfer that automatically decrements the ECX register before testing ECX for the branch condition. If ECX is non-zero, the program branches to the target label specified in the instruction. The LOOP instruction causes the repetition of a code section

until the operation of the LOOP instruction decrements ECX to a value of zero. If LOOP finds ECX=0, control transfers to the instruction immediately following the LOOP instruction. If the value of ECX is initially zero, then the LOOP executes $2^{32}$ times.

LOOPE (Loop While Equal) and LOOPZ (Loop While Zero) are synonyms for the same instruction. These instructions automatically decrement the ECX register before testing ECX and ZF for the branch conditions. If ECX is non-zero and ZF=1, the program branches to the target label specified in the instruction. If LOOPE or LOOPZ finds that ECX=0 or ZF=0, control transfers to the instruction immediately following the LOOPE or LOOPZ instruction.

LOOPNE (Loop While Not Equal) and LOOPNZ (Loop While Not Zero) are synonyms for the same instruction. These instructions automatically decrement the ECX register before testing ECX and ZF for the branch conditions. If ECX is non-zero and ZF=0, the program branches to the target label specified in the instruction. If LOOPNE or LOOPNZ finds that ECX=0 or ZF=1, control transfers to the instruction immediately following the LOOPNE or LOOPNZ instruction.

#### 3.5.2.3 Executing a Loop or Repeat Zero Times

JCXZ (Jump if ECX Zero) branches to the label specified in the instruction if it finds a value of zero in ECX. JCXZ is useful in combination with the LOOP instruction and with the string scan and compare instructions, all of which decrement ECX. Sometimes, it is desirable to design a loop that executes zero times if the count variable in ECX is initialized to zero. Because the LOOP instructions (and repeat prefixes) decrement ECX before they test it, a loop will execute $2^{32}$ times if the program enters the loop with a zero value in ECX. A programmer may conveniently overcome this problem with JCXZ, which enables the program to branch around the code within the loop if ECX is zero when JCXZ executes. When used with repeated string scan and compare instructions, JCXZ can determine whether the repetitions terminated due to zero in ECX or due to satisfaction of the scan or compare conditions.

### 3.5.3 Software-Generated Interrupts

The INT n, INTO, and BOUND instructions allow the programmer to specify a transfer to an interrupt service routine from within a program.

INT n (Software Interrupt) activates the interrupt service routine that corresponds to the number coded within the instruction. The INT instruction may specify any interrupt type. Programmers may use this flexibility to implement multiple types of internal interrupts or to test the operation of interrupt service routines. (Interrupts 0-31 are reserved by Intel.) The interrupt service routine terminates with an IRET instruction that returns control to the instruction that follows INT.

INTO (Interrupt on Overflow) invokes interrupt 4 if OF is set. Interrupt 4 is reserved for this purpose. OF is set by several arithmetic, logical, and string instructions.

BOUND (Detect Value Out of Range) verifies that the signed value contained in the specified register lies within specified limits. An interrupt (INT 5) occurs if the value contained in the register is less than the lower bound or greater than the upper bound.

The BOUND instruction includes two operands. The first operand specifies the register being tested. The second operand contains the effective relative address of the two signed BOUND limit values. The BOUND instruction assumes that the upper limit and lower limit are in adjacent memory locations. These limit values cannot be register operands; if they are, an invalid opcode exception occurs.

BOUND is useful for checking array bounds before using a new index value to access an element within the array. BOUND provides a simple way to check the value of an index register before the program overwrites information in a location beyond the limit of the array.

The block of memory that specifies the lower and upper limits of an array might typically reside just before the array itself. This makes the array bounds accessible at a constant offset from the beginning of the array. Because the address of the array will already be present in a register, this practice avoids extra calculations to obtain the effective address of the array bounds.

The upper and lower limit values may each be a word or a doubleword.

## 3.6 String and Character Translation Instructions

The instructions in this category operate on strings rather than on logical or numeric values. Refer also to the section on I/O for information about the string I/O instructions (also known as block I/O).

The power of 80386 string operations derives from the following features of the architecture:

1. A set of primitive string operations

   MOVS —— Move String
   CMPS —— Compare string
   SCAS —— Scan string
   LODS —— Load string
   STOS —— Store string

2. Indirect, indexed addressing, with automatic incrementing or decrementing of the indexes.

   Indexes:

      ESI —— Source index register
      EDI —— Destination index register

   Control flag:

      DF —— Direction flag

Control flag instructions:

```
CLD    —— Clear direction flag instruction
STD    —— Set direction flag instruction
```

3. Repeat prefixes

```
REP          —— Repeat while ECX not xero
REPE/REPZ    —— Repeat while equal or zero
REPNE/REPNZ  —— Repeat while not equal or not zero
```

The primitive string operations operate on one element of a string. A string element may be a byte, a word, or a doubleword. The string elements are addressed by the registers ESI and EDI. After every primitive operation ESI and/or EDI are automatically updated to point to the next element of the string. If the direction flag is zero, the index registers are incremented; if one, they are decremented. The amount of the increment or decrement is 1, 2, or 4 depending on the size of the string element.

### 3.6.1 Repeat Prefixes

The repeat prefixes REP (Repeat While ECX Not Zero), REPE/REPZ (Repeat While Equal/Zero), and REPNE/REPNZ (Repeat While Not Equal/Not Zero) specify repeated operation of a string primitive. This form of iteration allows the CPU to process strings much faster than would be possible with a regular software loop.

When a primitive string operation has a repeat prefix, the operation is executed repeatedly, each time using a different element of the string. The repetition terminates when one of the conditions specified by the prefix is satisfied.

At each repetition of the primitive instruction, the string operation may be suspended temporarily in order to handle an exception or external interrupt. After the interruption, the string operation can be restarted again where it left off. This method of handling strings allows operations on strings of arbitrary length, without affecting interrupt response.

All three prefixes causes the hardware to automatically repeat the associated string primitive until ECX=0. The differences among the repeat prefixes have to do with the second termination condition. REPE/REPZ and REPNE/REPNZ are used exclusively with the SCAS (Scan String) and CMPS (Compare String) primitives. When these prefixes are used, repetition of the next instruction depends on the zero flag (ZF) as well as the ECX register. ZF does not require initialization before execution of a repeated string instruction, because both SCAS and CMPS set ZF according to the results of the comparisons they make. The differences are summarized in the accompanying table.

| Prefix | Termination Condition 1 | Termination Condition 2 |
|---|---|---|
| REP | ECX = 0 | (none) |
| REPE/REPZ | ECX = 0 | ZF = 0 |
| REPNE/REPNZ | ECX = 0 | ZF = 1 |

### 3.6.2 Indexing and Direction Flag Control

The addresses of the operands of string primitives are determined by the ESI and EDI registers. ESI points to source operands. By default, ESI refers to a location in the segment indicated by the DS segment register. A segment-override prefix may be used, however, to cause ESI to refer to CS, SS, ES, FS, or GS. EDI points to destination operands in the segment indicated by ES; no segment override is possible. The use of two different segment registers in one instruction allows movement of strings between different segments.

This use of ESI and DSI has led to the descriptive names source index and destination index for the ESI and EDI registers, respectively. In all cases other than string instructions, however, the ESI and EDI registers may be used as general-purpose registers.

When ESI and EDI are used in string primitives, they are automatically incremented or decremented after to operation. The direction flag determines whether they are incremented or decremented. The instruction CLD puts zero in DF, causing the index registers to be incremented; the instruction STD puts one in DF, causing the index registers to be decremented. Programmers should always put a known value in DF before using string instructions in a procedure.

### 3.6.3 String Instructions

MOVS (Move String) moves the string element pointed to by ESI to the location pointed to by EDI. MOVSB operates on byte elements, MOVSW operates on word elements, and MOVSD operates on doublewords. The destination segment register cannot be overridden by a segment override prefix, but the source segment register can be overridden.

The MOVS instruction, when accompanied by the REP prefix, operates as a memory-to-memory block transfer. To set up for this operation, the program must initialize ECX and the register pairs ESI and EDI. ECX specifies the number of bytes, words, or doublewords in the block.

If DF=0, the program must point ESI to the first element of the source string and point EDI to the destination address for the first element. If DF=1, the program must point these two registers to the last element of the source string and to the destination address for the last element, respectively.

CMPS (Compare Strings) subtracts the destination string element (at ES:EDI) from the source string element (at ESI) and updates the flags AF, SF, PF, CF and OF. If the string elements are equal, ZF=1; otherwise, ZF=0. If DF=0, the processor increments the memory pointers (ESI and EDI) for the two strings. CMPSB compares bytes, CMPSW compares words, and CMPSD compares doublewords. The segment register used for the source address can be changed with a segment override prefix while the destination segment register cannot be overridden.

SCAS (Scan String) subtracts the destination string element at ES:EDI from EAX, AX, or AL and updates the flags AF, SF, ZF, PF, CF and OF. If the

values are equal, ZF=1; otherwise, ZF=0. If DF=0, the processor increments the memory pointer (EDI) for the string. SCASB scans bytes; SCASW scans words; SCASD scans doublewords. The destination segment register (ES) cannot be overridden.

When either the REPE or REPNE prefix modifies either the SCAS or CMPS primitives, the processor compares the value of the current string element with the value in EAX for doubleword elements, in AX for word elements, or in AL for byte elements. Termination of the repeated operation depends on the resulting state of ZF as well as on the value in ECX.

LODS (Load String) places the source string element at ESI into EAX for doubleword strings, into AX for word strings, or into AL for byte strings. LODS increments or decrements ESI according to DF.

STOS (Store String) places the source string element from EAX, AX, or AL into the string at ES:DSI. STOS increments or decrements EDI according to DF.

## 3.7 Instructions for Block-Structured Languages

The instructions in this section provide machine-language support for functions normally found in high-level languages. These instructions include ENTER and LEAVE, which simplify the programming of procedures.

ENTER (Enter Procedure) creates a stack frame that may be used to implement the scope rules of block-structured high-level languages. A LEAVE instruction at the end of a procedure complements an ENTER at the beginning of the procedure to simplify stack management and to control access to variables for nested procedures.

The ENTER instruction includes two parameters. The first parameter specifies the number of bytes of dynamic storage to be allocated on the stack for the routine being entered. The second parameter corresponds to the lexical nesting level (0-31) of the routine. (Note that the lexical level has no relationship to either the protection privilege levels or to the I/O privilege level.)

The specified lexical level determines how many sets of stack frame pointers the CPU copies into the new stack frame from the preceding frame. This list of stack frame pointers is sometimes called the display. The first word of the display is a pointer to the last stack frame. This pointer enables a LEAVE instruction to reverse the action of the previous ENTER instruction by effectively discarding the last stack frame.

```
Example: ENTER 2048,3
```

Allocates 2048 bytes of dynamic storage on the stack and sets up pointers to two previous stack frames in the stack frame that ENTER creates for this procedure.

After ENTER creates the new display for a procedure, it allocates the dynamic storage space for that procedure by decrementing ESP by the number of bytes specified in the first parameter. This new value of ESP serves as a starting point for all PUSH and POP operations within that procedure.

To enable a procedure to address its display, ENTER leaves EBP pointing to the beginning of the new stack frame. Data manipulation instructions that specify EBP as a base register implicitly address locations within the stack segment instead of the data segment.

The ENTER instruction can be used in two ways: nested and non-nested. If the lexical level is 0, the non-nested form is used. Since the second operand is 0, ENTER pushes EBP, copies ESP to EBP and then subtracts the first operand from ESP. The nested form of ENTER occurs when the second parameter (lexical level) is not 0.

Figure 3-16 gives the formal definition of ENTER.

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

A program operating at a higher lexical level calling a program at a lower lexical level requires that the called procedure should have access to the variables of the calling program. ENTER provides this access through a display that provides addressability to the calling program's stack frame.

A procedure calling another procedure at the same lexical level implies that they are parallel procedures and that the called procedure should not have access to the variables of the calling procedure. In this case, ENTER copies only that portion of the display from the calling procedure which refers to previously nested procedures operating at higher lexical levels. The new stack frame does not include the pointer for addressing the calling procedure's stack frame.

ENTER treats a reentrant procedure as a procedure calling another procedure at the same lexical level. In this case, each succeeding iteration of the reentrant procedure can address only its own variables and the variables of the calling procedures at higher lexical levels. A reentrant procedure can always address its own variables; it does not require pointers to the stack frames of previous iterations.

By copying only the stack frame pointers of procedures at higher lexical levels, ENTER makes sure that procedures access only those variables of higher lexical levels, not those at parallel lexical levels (see Figure 3-17). Figures 3-18 through 3-21 demonstrate the actions of the ENTER instruction if the modules shown in Figure 3-17 were to call one another in alphabetic order.

Block-structured high-level languages can use the lexical levels defined by ENTER to control access to the variables of previously nested procedures. Referring to Figure 3-17 for example, if PROCEDURE A calls PROCEDURE B which, in turn, calls PROCEDURE C, then PROCEDURE C will have access to the variables of MAIN and PROCEDURE A, but not PROCEDURE B because they operate at the same lexical level. Following is the complete definition of access to variables for Figure 3-17.

1. MAIN PROGRAM has variables at fixed locations.

2. PROCEDURE A can access only the fixed variables of MAIN.

3. PROCEDURE B can access only the variables of PROCEDURE A and MAIN. PROCEDURE B cannot access the variables of PROCEDURE C or PROCEDURE D.

4. PROCEDURE C can access only the variables of PROCEDURE A and MAIN. PROCEDURE C cannot access the variables of PROCEDURE B or PROCEDURE D.

5. PROCEDURE D can access the variables of PROCEDURE C, PROCEDURE A, and MAIN. PROCEDURE D cannot access the variables of PROCEDURE B.

ENTER at the beginning of the MAIN PROGRAM creates dynamic storage space for MAIN but copies no pointers. The first and only word in the display points to itself because there is no previous value for LEAVE to return to EBP. See Figure 3-18.

After MAIN calls PROCEDURE A, ENTER creates a new display for PROCEDURE A with the first word pointing to the previous value of EBP (BPM for LEAVE to return to the MAIN stack frame) and the second word pointing to the current value of EBP. Procedure A can access variables in MAIN since MAIN is at level 1. Therefore the base for the dynamic storage for MAIN is at [EBP-2]. All dynamic variables for MAIN are at a fixed offset from this value. See Figure 3-19.

After PROCEDURE A calls PROCEDURE B, ENTER creates a new display for PROCEDURE B with the first word pointing to the previous value of EBP, the second word pointing to the value of EBP for MAIN, and the third word pointing to the value of EBP for A and the last word pointing to the current EBP. B can access variables in A and MAIN by fetching from the display the base addresses of the respective dynamic storage areas. See Figure 3-20.

After PROCEDURE B calls PROCEDURE C, ENTER creates a new display for PROCEDURE C with the first word pointing to the previous value of EBP, the second word pointing to the value of EBP for MAIN, and the third word pointing to the EBP value for A and the third word pointing to the current value of EBP. Because PROCEDURE B and PROCEDURE C have the same lexical level, PROCEDURE C is not allowed access to variables in B and therefore does not receive a pointer to the beginning of PROCEDURE B's stack frame. See Figure 3-21.

LEAVE (Leave Procedure) reverses the action of the previous ENTER instruction. The LEAVE instruction does not include any operands. LEAVE copies EBP to ESP to release all stack space allocated to the procedure by the most recent ENTER instruction. Then LEAVE pops the old value of EBP from the stack. A subsequent RET instruction can then remove any arguments that were pushed on the stack by the calling program for use by the called procedure.

**Figure 3-16. Formal Definition of the ENTER Instruction**

The formal definition of the ENTER instruction for all cases is given by the following listing. LEVEL denotes the value of the second operand.

```
Push EBP
Set a temporary value FRAME_PTR := ESP
If LEVEL > 0 then
      Repeat (LEVEL-1) times:
          EBP :=EBP - 4
          Push the doubleword pointed to by EBP
      End repeat
      Push FRAME_PTR
End if
EBP := FRAME_PTR
ESP := ESP - first operand.
```

**Figure 3-17. Variable Access in Nested Procedures**

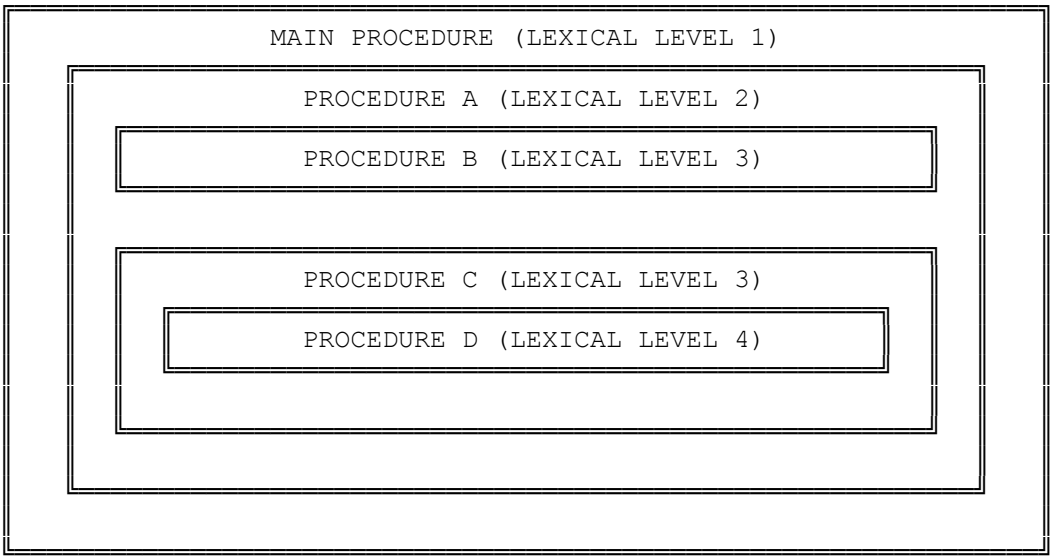



**Figure 3-18. Stack Frame for MAIN at Level 1**

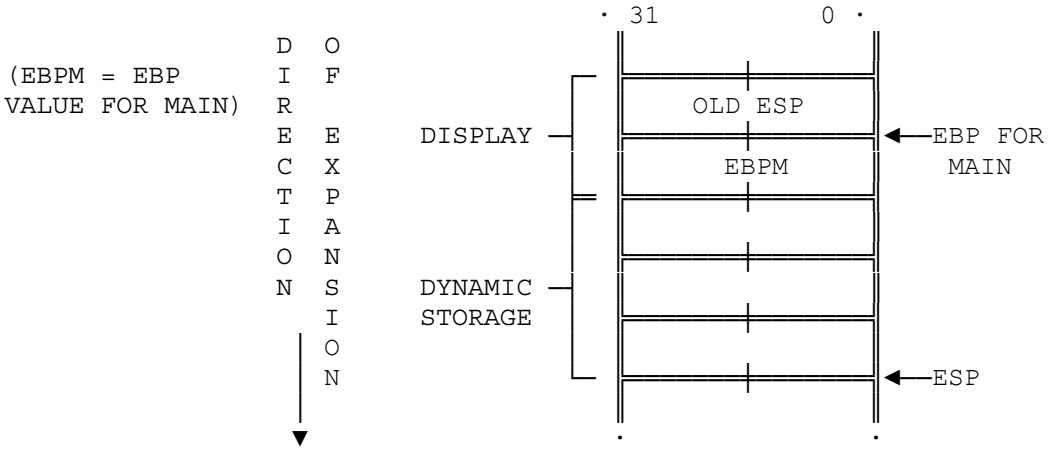


**Figure 3-19. Stack Frame for Procedure A**

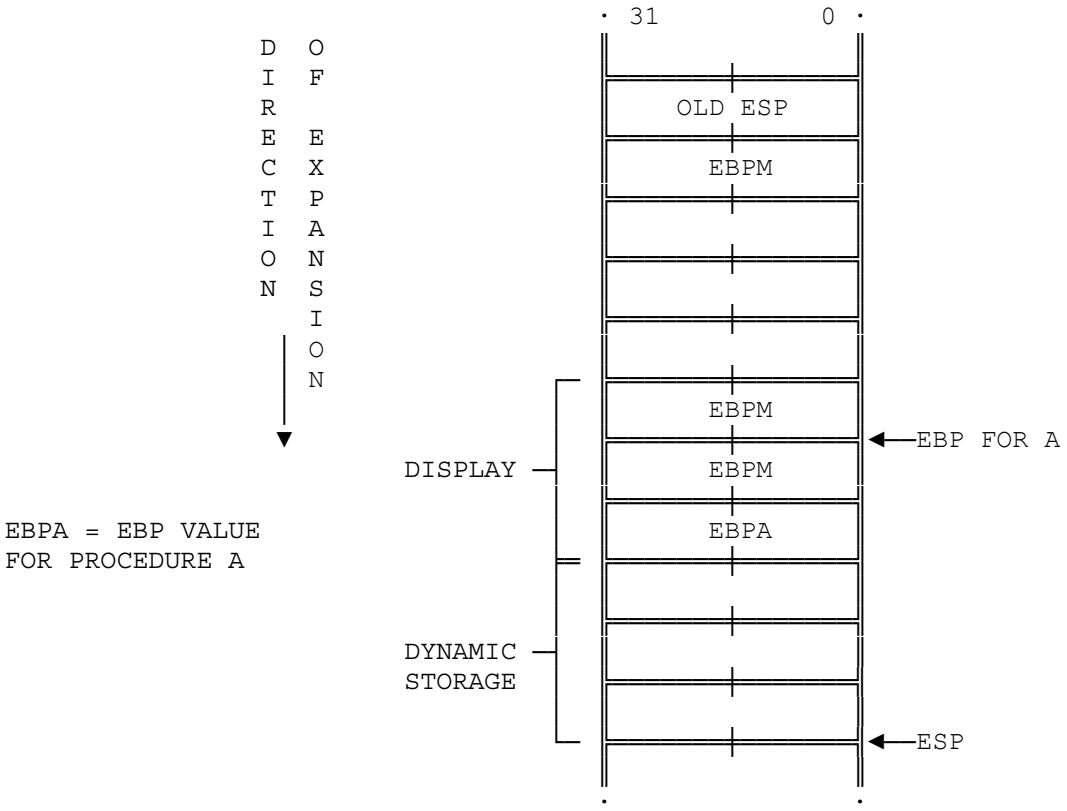


**Figure 3-20. Stack Frame for Procedure B at Level 3 Called from A**

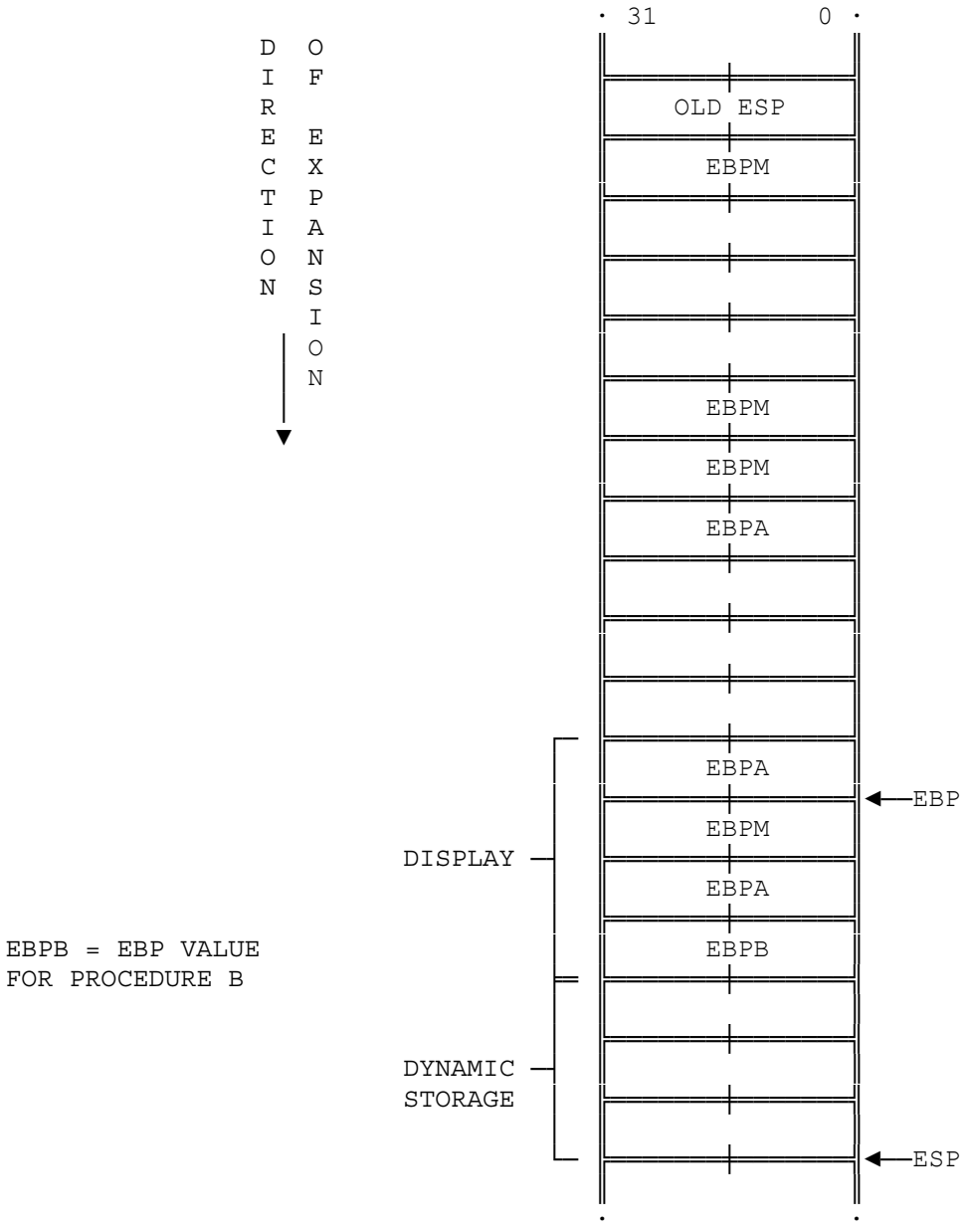
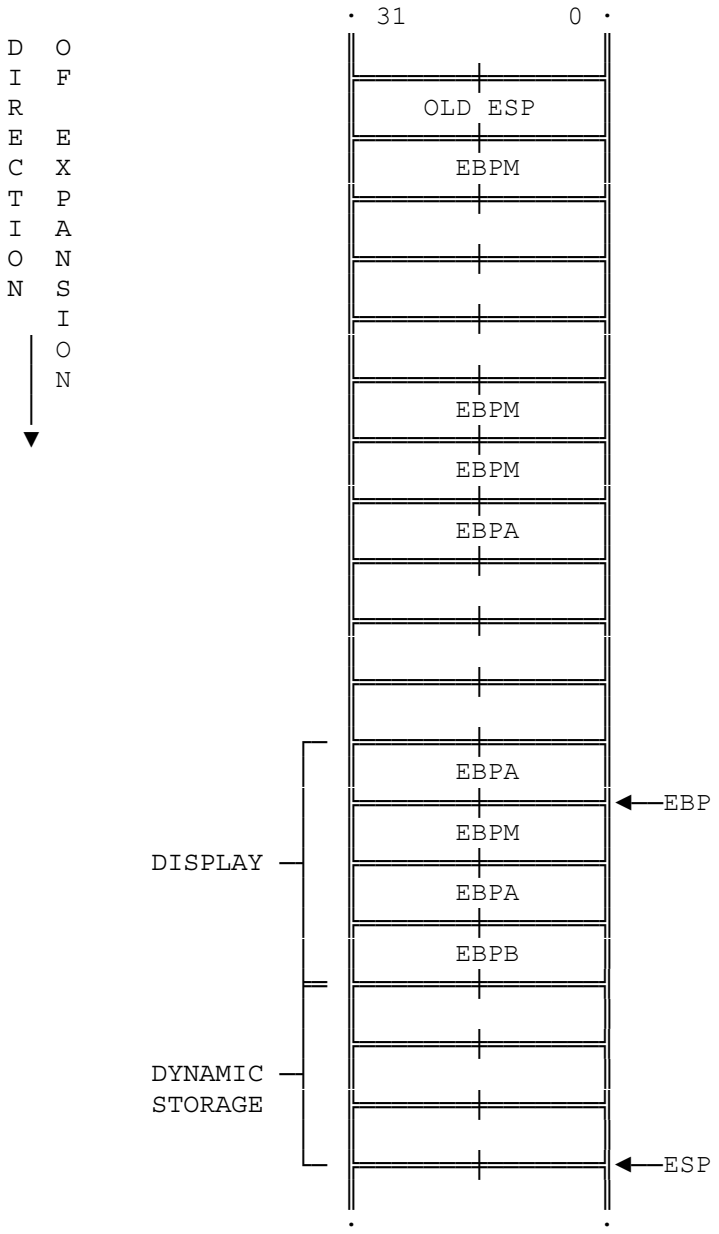


**Figure 3-21. Stack Frame for Procedure C at Level 3 Called from B**



## 3.8 Flag Control Instructions

The flag control instructions provide a method for directly changing the state of bits in the flag register.

### 3.8.1 Carry and Direction Flag Control Instructions

The carry flag instructions are useful in conjunction with rotate-with-carry instructions RCL and RCR. They can initialize the carry flag, CF, to a known state before execution of a rotate that moves the carry bit into one end of the rotated operand.

The direction flag control instructions are specifically included to set or clear the direction flag, DF, which controls the left-to-right or right-to-left direction of string processing. If DF=0, the processor automatically increments the string index registers, ESI and EDI, after each execution of a string primitive. If DF=1, the processor decrements these index registers. Programmers should use one of these instructions before any procedure that uses string instructions to insure that DF is set properly.

| Flag Control Instruction | Effect |
|---|---|
| STC (Set Carry Flag) | CF ← 1 |
| CLC (Clear Carry Flag) | CF ← 0 |
| CMC (Complement Carry Flag) | CF ← NOT (CF) |
| CLD (Clear Direction Flag) | DF ← 0 |
| STD (Set Direction Flag) | DF ← 1 |

### 3.8.2 Flag Transfer Instructions

Though specific instructions exist to alter CF and DF, there is no direct method of altering the other applications-oriented flags. The flag transfer instructions allow a program to alter the other flag bits with the bit manipulation instructions after transferring these flags to the stack or the AH register.

The instructions LAHF and SAHF deal with five of the status flags, which are used primarily by the arithmetic and logical instructions.

LAHF (Load AH from Flags) copies SF, ZF, AF, PF, and CF to AH bits 7, 6, 4, 2, and 0, respectively (see Figure 3-22). The contents of the remaining bits (5, 3, and 1) are undefined. The flags remain unaffected.

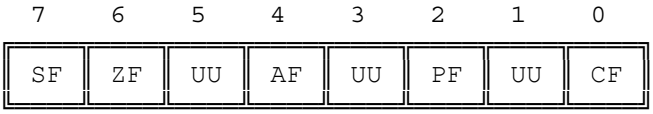
SAHF (Store AH into Flags) transfers bits 7, 6, 4, 2, and 0 from AH into SF, ZF, AF, PF, and CF, respectively (see Figure 3-22).

The PUSHF and POPF instructions are not only useful for storing the flags in memory where they can be examined and modified but are also useful for preserving the state of the flags register while executing a procedure.

PUSHF (Push Flags) decrements ESP by two and then transfers the low-order word of the flags register to the word at the top of stack pointed to by ESP (see Figure 3-23). The variant PUSHFD decrements ESP by four, then transfers both words of the extended flags register to the top of the stack pointed to by ESP (the VM and RF flags are not moved, however).

POPF (Pop Flags) transfers specific bits from the word at the top of stack into the low-order byte of the flag register (see Figure 3-23), then increments ESP by two. The variant POPFD transfers specific bits from the doubleword at the top of the stack into the extended flags register (the RF and VM flags are not changed, however), then increments ESP by four.

Figure 3-22. LAHF and SAHF




LAHF LOADS FIVE FLAGS FROM THE FLAG REGISTER INTO REGISTER AH. SAHF STORES THESE SAME FIVE FLAGS FROM AH INTO THE FLAG REGISTER. THE BIT POSITION OF EACH FLAG IS THE SAME IN AH AS IT IS IN THE FLAG REGISTER. THE REMAINING BITS (MARKED UU) ARE RESERVED; DO NOT DEFINE.

## 3.9 Coprocessor Interface Instructions

A numerics coprocessor (e.g., the 80387 or 80287) provides an extension to the instruction set of the base architecture. The coprocessor extends the instruction set of the base architecture to support high-precision integer and floating-point calculations. This extended instruction set includes arithmetic, comparison, transcendental, and data transfer instructions. The coprocessor also contains a set of useful constants to enhance the speed of numeric calculations.

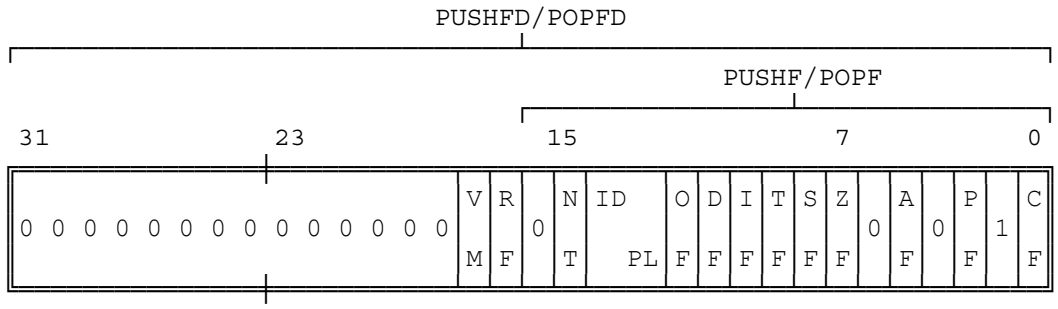
A program contains instructions for the coprocessor in line with the instructions for the CPU. The system executes these instructions in the same order as they appear in the instruction stream. The coprocessor operates concurrently with the CPU to provide maximum throughput for numeric calculations.

The 80386 also has features to support emulation of the numerics coprocessor when the coprocessor is absent. The software emulation of the coprocessor is transparent to application software but requires more time for execution. Refer to Chapter 11 for more information on coprocessor emulation.

ESC (Escape) is a 5-bit sequence that begins the opcodes that identify floating point numeric instructions. The ESC pattern tells the 80386 to send the opcode and addresses of operands to the numerics coprocessor. The numerics coprocessor uses the escape instructions to perform high-performance, high-precision floating point arithmetic that conforms to the IEEE floating point standard 754.

WAIT (Wait) is an 80386 instruction that suspends program execution until the 80386 CPU detects that the BUSY pin is inactive. This condition indicates that the coprocessor has completed its processing task and that the CPU may obtain the results.

**Figure 3-23. Flag Format for PUSHF and POPF**




BITS MARKED 0 AND 1 ARE RESERVED BY INTEL. DO NOT DEFINE.

SYSTEMS FLAGS (INCLUDING THE IOPL FIELD, AND THE VM, RF, AND IF FLAGS) ARE PUSHED AND ARE VISIBLE TO APPLICATIONS PROGRAMS. HOWEVER, WHEN AN APPLICATIONS PROGRAM POPS THE FLAGS, THESE ITEMS ARE NOT CHANGED, REGARDLESS OF THE VALUES POPPED INTO THEM.

## 3.10 Segment Register Instructions

This category actually includes several distinct types of instructions. These various types are grouped together here because, if systems designers choose an unsegmented model of memory organization, none of these instructions is used by applications programmers. The instructions that deal with segment registers are:

1. Segment-register transfer instructions.

```
MOV SegReg, ...
MOV ..., SegReg
PUSH SegReg
POP SegReg
```

2. Control transfers to another executable segment.

```
JMP far    ; direct and indirect
CALL far
RET far
```

3. Data pointer instructions.

```
LDS
LES
LFS
LGS
LSS
```

Note that the following interrupt-related instructions are different; all are capable of transferring control to another segment, but the use of segmentation is not apparent to the applications programmer.

INT n
INTO
BOUND
IRET

### 3.10.1 Segment-Register Transfer Instructions

The MOV, POP, and PUSH instructions also serve to load and store segment registers. These variants operate similarly to their general-register counterparts except that one operand can be a segment register. MOV cannot move segment register to a segment register. Neither POP nor MOV can place a value in the code-segment register CS; only the far control-transfer instructions can change CS.

### 3.10.2 Far Control Transfer Instructions

The far control-transfer instructions transfer control to a location in another segment by changing the content of the CS register.

Direct far JMP. Direct JMP instructions that specify a target location outside the current code segment contain a far pointer. This pointer consists of a selector for the new code segment and an offset within the new segment.

Indirect far JMP. Indirect JMP instructions that specify a target location outside the current code segment use a 48-bit variable to specify the far pointer.

Far CALL. An intersegment CALL places both the value of EIP and CS on the stack.

Far RET. An intersegment RET restores the values of both CS and EIP which were saved on the stack by the previous intersegment CALL instruction.

### 3.10.3 Data Pointer Instructions

The data pointer instructions load a pointer (consisting of a segment selector and an offset) to a segment register and a general register.

LDS (Load Pointer Using DS) transfers a pointer variable from the source operand to DS and the destination register. The source operand must be a memory operand, and the destination operand must be a general register. DS receives the segment-selector of the pointer. The destination register receives the offset part of the pointer, which points to a specific location within the segment.

Example: LDS ESI, STRING_X

Loads DS with the selector identifying the segment pointed to by a STRING_X, and loads the offset of STRING_X into ESI. Specifying ESI as the destination operand is a convenient way to prepare for a string operation on a source string that is not in the current data segment.

LES (Load Pointer Using ES) operates identically to LDS except that ES receives the segment selector rather than DS.

Example: LES EDI, DESTINATION_X

Loads ES with the selector identifying the segment pointed to by DESTINATION_X, and loads the offset of DESTINATION_X into EDI. This instruction provides a convenient way to select a destination for a string operation if the desired location is not in the current extra segment.

LFS (Load Pointer Using FS) operates identically to LDS except that FS receives the segment selector rather than DS.

LGS (Load Pointer Using GS) operates identically to LDS except that GS receives the segment selector rather than DS.

LSS (Load Pointer Using SS) operates identically to LDS except that SS receives the segment selector rather than DS. This instruction is especially important, because it allows the two registers that identify the stack (SS:ESP) to be changed in one uninterruptible operation. Unlike the other instructions which load SS, interrupts are not inhibited at the end of the LSS instruction. The other instructions (e.g., POP SS) inhibit interrupts to permit the following instruction to load ESP, thereby forming an indivisible load of SS:ESP. Since both SS and ESP can be loaded by LSS, there is no need to inhibit interrupts.

## 3.11 Miscellaneous Instructions

The following instructions do not fit in any of the previous categories, but are nonetheless useful.

### 3.11.1 Address Calculation Instruction

LEA (Load Effective Address) transfers the offset of the source operand (rather than its value) to the destination operand. The source operand must be a memory operand, and the destination operand must be a general register. This instruction is especially useful for initializing registers before the execution of the string primitives (ESI, EDI) or the XLAT instruction (EBX). The LEA can perform any indexing or scaling that may be needed.

Example: LEA EBX, EBCDIC_TABLE

Causes the processor to place the address of the starting location of the table labeled EBCDIC_TABLE into EBX.

### 3.11.2 No-Operation Instruction

NOP (No Operation) occupies a byte of storage but affects nothing but the instruction pointer, EIP.

### 3.11.3 Translate Instruction

XLAT (Translate) replaced a byte in the AL register with a byte from a user-coded translation table. When XLAT is executed, AL should have the unsigned index to the table addressed by EBX. XLAT changes the contents of AL from table index to table entry. EBX is unchanged. The XLAT instruction is useful for translating from one coding system to another such as from ASCII to EBCDIC. The translate table may be up to 256 bytes long. The value placed in the AL register serves as an index to the location of the corresponding translation value.

# PART II SYSTEMS PROGRAMMING

# Chapter 4 Systems Architecture

---

Many of the architectural features of the 80386 are used only by systems programmers. This chapter presents an overview of these aspects of the architecture.

The systems-level features of the 80386 architecture include:

- Memory Management
- Protection
- Multitasking
- Input/Output
- Exceptions and Interrupts
- Initialization
- Coprocessing and Multiprocessing
- Debugging

These features are implemented by registers and instructions, all of which are introduced in the following sections. The purpose of this chapter is not to explain each feature in detail, but rather to place the remaining chapters of Part II in perspective. Each mention in this chapter of a register or instruction is either accompanied by an explanation or a reference to a following chapter where detailed information can be obtained.

## 4.1 Systems Registers

The registers designed for use by systems programmers fall into these classes:

- EFLAGS
- Memory-Management Registers
- Control Registers
- Debug Registers
- Test Registers

### 4.1.1 Systems Flags

The systems flags of the EFLAGS register control I/O, maskable interrupts, debugging, task switching, and enabling of virtual 8086 execution in a protected, multitasking environment. These flags are highlighted in Figure 4-1.

IF (Interrupt-Enable Flag, bit 9)

Setting IF allows the CPU to recognize external (maskable) interrupt requests. Clearing IF disables these interrupts. IF has no effect on either exceptions or nonmaskable external interrupts. Refer to Chapter 9 for more details about interrupts.

NT (Nested Task, bit 14)

The processor uses the nested task flag to control chaining of interrupted and called tasks. NT influences the operation of the IRET instruction. Refer to Chapter 7 and Chapter 9 for more information on nested tasks.

RF (Resume Flag, bit 16)

The RF flag temporarily disables debug exceptions so that an instruction can be restarted after a debug exception without immediately causing another debug exception. Refer to Chapter 12 for details.

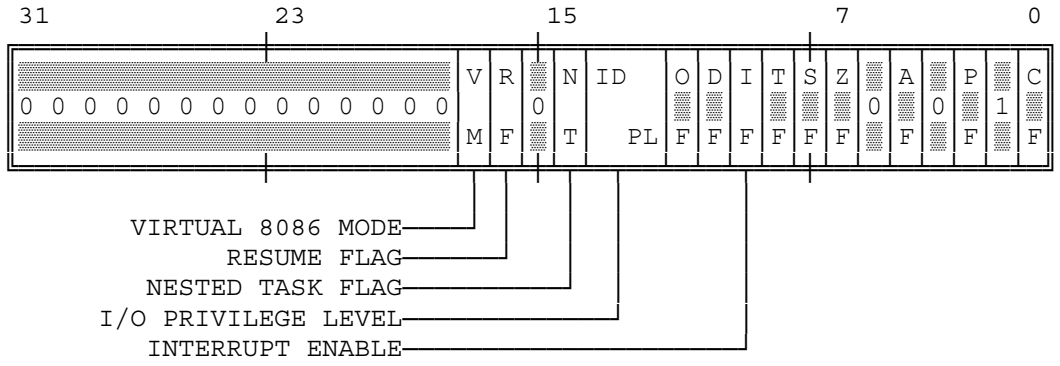
TF (Trap Flag, bit 8)

Setting TF puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an exception after each instruction, allowing a program to be inspected as it executes each instruction. Single-stepping is just one of several debugging features of the 80386. Refer to Chapter 12 for additional information.

VM (Virtual 8086 Mode, bit 17)

When set, the VM flag indicates that the task is executing an 8086 program. Refer to Chapter 14 for a detailed discussion of how the 80386 executes 8086 tasks in a protected, multitasking environment.

**Figure 4-1. System Flags of EFLAGS Register**




NOTE
0 OR 1 INDICATES INTEL RESERVED. DO NOT DEFINE.

### 4.1.2 Memory-Management Registers

Four registers of the 80386 locate the data structures that control segmented memory management:

GDTR Global Descriptor Table Register
LDTR Local Descriptor Table Register

These registers point to the segment descriptor tables GDT and LDT. Refer to Chapter 5 for an explanation of addressing via descriptor tables.

IDTR Interrupt Descriptor Table Register

This register points to a table of entry points for interrupt handlers (the IDT). Refer to Chapter 9 for details of the interrupt mechanism.

TR Task Register

This register points to the information needed by the processor to define the current task. Refer to Chapter 7 for a description of the multitasking features of the 80386.

### 4.1.3 Control Registers

Figure 4-2 shows the format of the 80386 control registers CR0, CR2, and CR3. These registers are accessible to systems programmers only via variants of the MOV instruction, which allow them to be loaded from or stored in general registers; for example:

```
MOV EAX, CR0
MOV CR3, EBX
```

CR0 contains system control flags, which control or indicate conditions that apply to the system as a whole, not to an individual task.

EM (Emulation, bit 2)

EM indicates whether coprocessor functions are to be emulated. Refer to Chapter 11 for details.

ET (Extension Type, bit 4)

ET indicates the type of coprocessor present in the system (80287 or 80387). Refer to Chapter 11 and Chapter 10 for details.

MP (Math Present, bit 1)

MP controls the function of the WAIT instruction, which is used to coordinate a coprocessor. Refer to Chapter 11 for details.

PE (Protection Enable, bit 0)

Setting PE causes the processor to begin executing in protected mode. Resetting PE returns to real-address mode. Refer to Chapter 14 and Chapter 10 for more information on changing processor modes.

PG (Paging, bit 31)

PG indicates whether the processor uses page tables to translate linear addresses into physical addresses. Refer to Chapter 5 for a description of page translation; refer to Chapter 10 for a discussion of how to set PG.

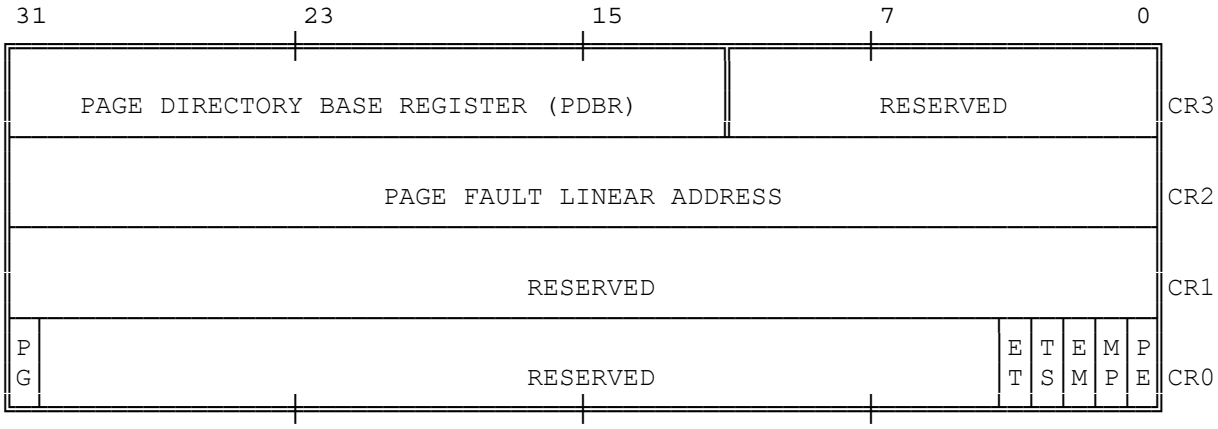
TS (Task Switched, bit 3)

The processor sets TS with every task switch and tests TS when interpreting coprocessor instructions. Refer to Chapter 11 for details.

CR2 is used for handling page faults when PG is set. The processor stores in CR2 the linear address that triggers the fault. Refer to Chapter 9 for a description of page-fault handling.

CR3 is used when PG is set. CR3 enables the processor to locate the page table directory for the current task. Refer to Chapter 5 for a description of page tables and page translation.

**Figure 4-2. Control Registers**




## 4.1.4 Debug Register

The debug registers bring advanced debugging abilities to the 80386, including data breakpoints and the ability to set instruction breakpoints without modifying code segments. Refer to Chapter 12 for a complete description of formats and usage.

### 4.1.5 Test Registers

The test registers are not a standard part of the 80386 architecture. They are provided solely to enable confidence testing of the translation lookaside buffer (TLB), the cache used for storing information from page tables. Chapter 12 explains how to use these registers.

## 4.2 Systems Instructions

Systems instructions deal with such functions as:

1. Verification of pointer parameters (refer to Chapter 6):

| | |
|---|---|
| ARPL | —— Adjust RPL |
| LAR | —— Load Access Rights |
| LSL | —— Load Segment Limit |
| VERR | —— Verify for Reading |
| VERW | —— Verify for Writing |

2. Addressing descriptor tables (refer to Chaper 5):

| | |
|---|---|
| LLDT | —— Load LDT Register |
| SLDT | —— Store LDT Register |
| LGDT | —— Load GDT Register |
| SGDT | —— Store GDT Register |

3. Multitasking (refer to Chapter 7):

| | |
|---|---|
| LTR | —— Load Task Register |
| STR | —— Store Task Register |

4. Coprocessing and Multiprocessing (refer to Chapter 11):

| | |
|---|---|
| CLTS | —— Clear Task-Switched Flag |
| ESC | —— Escape instructions |
| WAIT | —— Wait until Coprocessor not Busy |
| LOCK | —— Assert Bus-Lock Signal |

5. Input and Output (refer to Chapter 8):

| | |
|---|---|
| IN | —— Input |
| OUT | —— Output |
| INS | —— Input String |
| OUTS | —— Output String |

6. Interrupt control (refer to Chapter 9):

| | |
|---|---|
| CLI | —— Clear Interrupt-Enable Flag |
| STI | —— Set Interrupt-Enable Flag |
| LIDT | —— Load IDT Register |
| SIDT | —— Store IDT Register |

7. Debugging (refer to Chapter 12):

| | |
|---|---|
| MOV | —— Move to and from debug registers |

8. TLB testing (refer to Chapter 10):

| | |
|---|---|
| MOV | —— Move to and from test registers |

9. System Control:

| | |
|---|---|
| SMSW | —— Set MSW |
| LMSW | —— Load MSW |
| HLT | —— Halt Processor |
| MOV | —— Move to and from control registers |

The instructions SMSW and LMSW are provided for compatibility with the 80286 processor. 80386 programs access the MSW in CR0 via variants of the MOV instruction. HLT stops the processor until receipt of an INTR or RESET signal.

In addition to the chapters cited above, detailed information about each of these instructions can be found in the instruction reference chapter, Chapter 17.

# Chapter 5 Memory Management

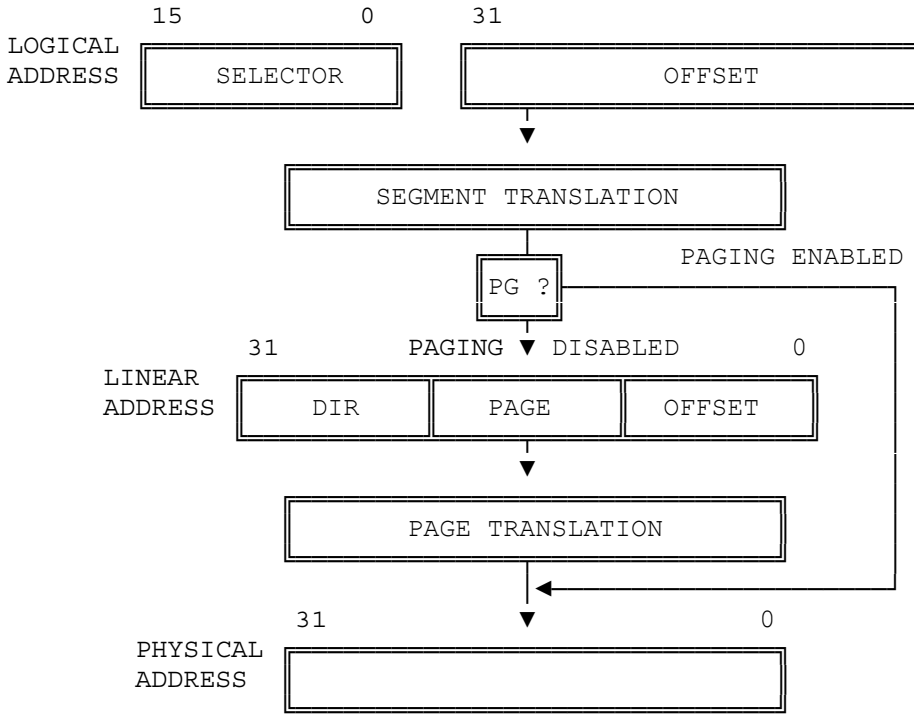
The 80386 transforms logical addresses (i.e., addresses as viewed by programmers) into physical address (i.e., actual addresses in physical memory) in two steps:

- Segment translation, in which a logical address (consisting of a segment selector and segment offset) are converted to a linear address.
- Page translation, in which a linear address is converted to a physical address. This step is optional, at the discretion of systems-software designers.

These translations are performed in a way that is not visible to applications programmers. Figure 5-1 illustrates the two translations at a high level of abstraction.

Figure 5-1 and the remainder of this chapter present a simplified view of the 80386 addressing mechanism. In reality, the addressing mechanism also includes memory protection features. For the sake of simplicity, however, the subject of protection is taken up in another chapter, Chapter 6.

**Figure 5-1. Address Translation Overview**



## 5.1 Segment Translation

Figure 5-2 shows in more detail how the processor converts a logical address into a linear address.

To perform this translation, the processor uses the following data structures:

- Descriptors
- Descriptor tables
- Selectors
- Segment Registers

### 5.1.1 Descriptors

The segment descriptor provides the processor with the data it needs to map a logical address into a linear address. Descriptors are created by compilers, linkers, loaders, or the operating system, not by applications programmers. Figure 5-3 illustrates the two general descriptor formats. All types of segment descriptors take one of these formats. Segment-descriptor fields are:

BASE: Defines the location of the segment within the 4 gigabyte linear address space. The processor concatenates the three fragments of the base address to form a single 32-bit value.

LIMIT: Defines the size of the segment. When the processor concatenates the two parts of the limit field, a 20-bit value results. The processor interprets the limit field in one of two ways, depending on the setting of the granularity bit:

1. In units of one byte, to define a limit of up to 1 megabyte.

2. In units of 4 Kilobytes, to define a limit of up to 4 gigabytes. The limit is shifted left by 12 bits when loaded, and low-order one-bits are inserted.

Granularity bit: Specifies the units with which the LIMIT field is interpreted. When thebit is clear, the limit is interpreted in units of one byte; when set, the limit is interpreted in units of 4 Kilobytes.

TYPE: Distinguishes between various kinds of descriptors.

DPL (Descriptor Privilege Level): Used by the protection mechanism (refer to Chapter 6).

Segment-Present bit: If this bit is zero, the descriptor is not valid for use in address transformation; the processor will signal an exception when a selector for the descriptor is loaded into a segment register. Figure 5-4 shows the format of a descriptor when the present-bit is zero. The operating system is free to use the locations marked AVAILABLE. Operating systems that implement segment-based virtual memory clear the present bit in either of these cases:

- When the linear space spanned by the segment is not mapped by the paging mechanism.
- When the segment is not present in memory.

Accessed bit: The processor sets this bit when the segment is accessed; i.e., a selector for the descriptor is loaded into a segment register or used by a selector test instruction. Operating systems that implement virtual memory at the segment level may, by periodically testing and clearing this bit, monitor frequency of segment usage.

Creation and maintenance of descriptors is the responsibility of systems software, usually requiring the cooperation of compilers, program loaders or system builders, and therating system.

**Figure 5-2. Segment Translation**

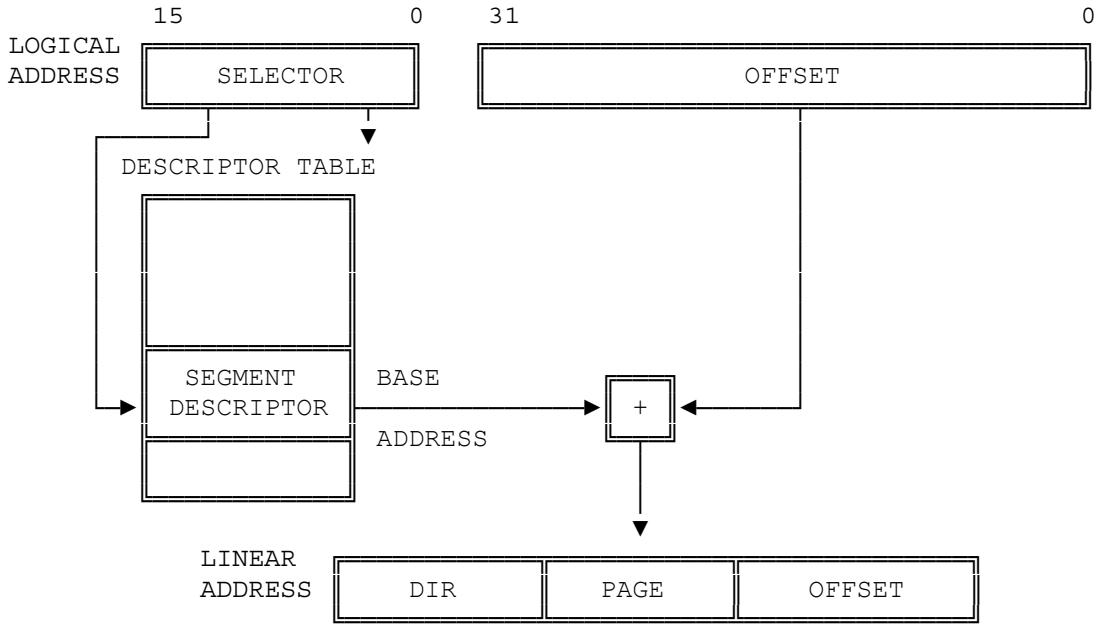
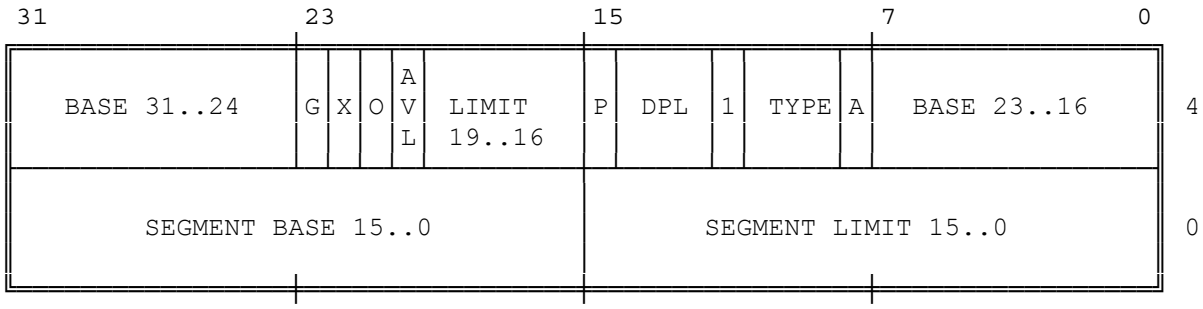
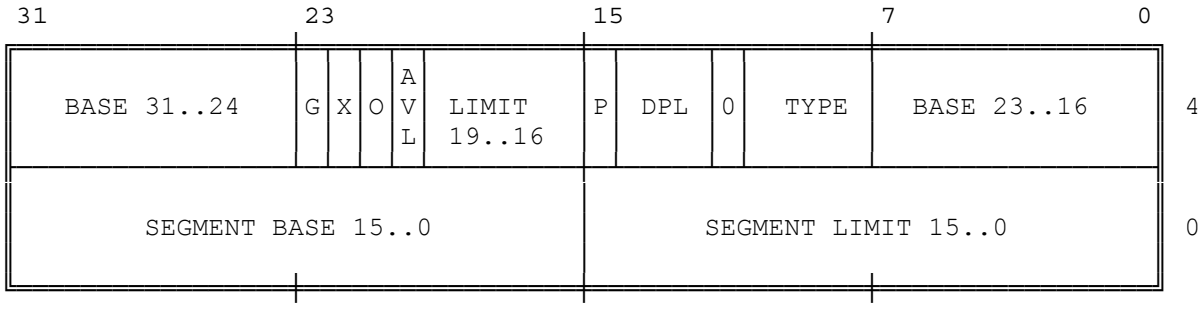


**Figure 5-3. General Segment-Descriptor Format**

DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS




DESCRIPTORS USED FOR SPECIAL SYSTEM SEGMENTS




A - ACCESSED
AVL - AVAILABLE FOR USE BY SYSTEMS PROGRAMMERS
DPL - DESCRIPTOR PRIVILEGE LEVEL
G - GRANULARITY
P - SEGMENT PRESENT

## 5.1.2 Descriptor Tables

Segment descriptors are stored in either of two kinds of descriptor table:

- The global descriptor table (GDT)
- A local descriptor table (LDT)

A descriptor table is simply a memory array of 8-byte entries that contain descriptors, as Figure 5-5 shows. A descriptor table is variable in length and may contain up to 8192 ($2^{13}$) descriptors. The first entry of the GDT (INDEX=0) is not used by the processor, however.

The processor locates the GDT and the current LDT in memory by means of the GDTR and LDTR registers. These registers store the base addresses of the tables in the linear address space and store the segment limits. The instructions LGDT and SGDT give access to the GDTR; the instructions LLDT and SLDT give access to the LDTR.

Figure 5-4. Format of Not-Present Descriptor

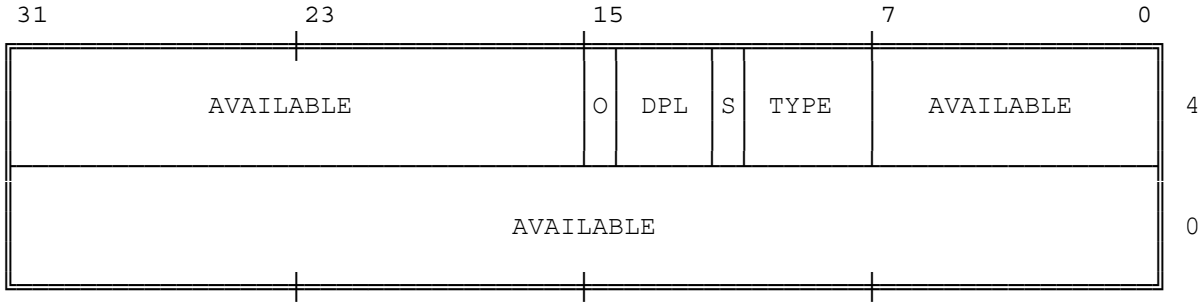
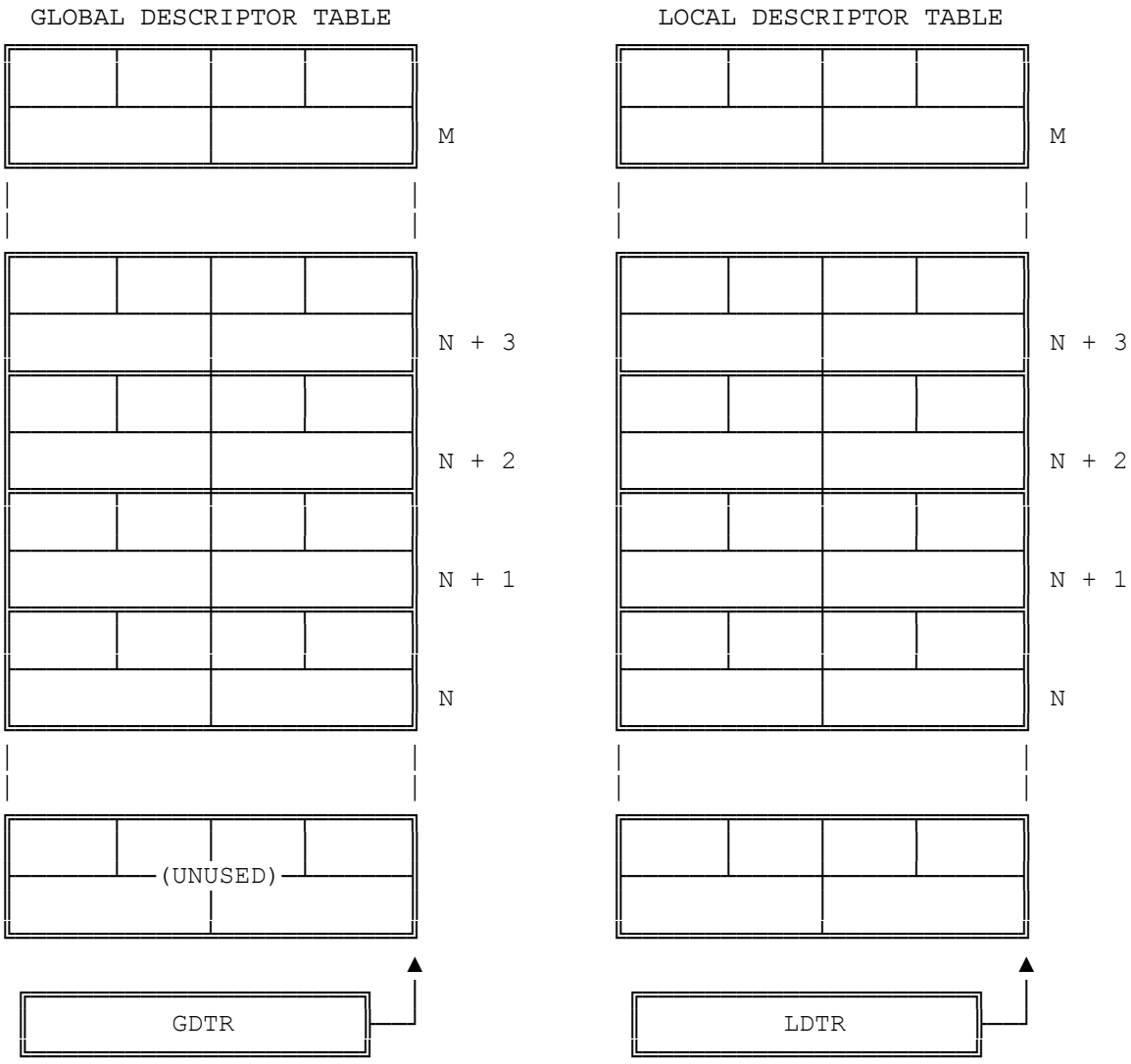



Figure 5-5. Descriptor Tables



### 5.1.3 Selectors

The selector portion of a logical address identifies a descriptor by specifying a descriptor table and indexing a descriptor within that table. Selectors may be visible to applications programs as a field within a pointer variable, but the values of selectors are usually assigned (fixed up) by linkers or linking loaders. Figure 5-6 shows the format of a selector.

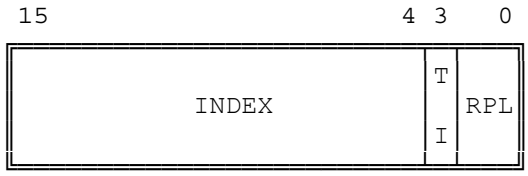
Index: Selects one of 8192 descriptors in a descriptor table. The processor simply multiplies this index value by 8 (the length of a descriptor), and adds the result to the base address of the descriptor table in order to access the appropriate segment descriptor in the table.

Table Indicator: Specifies to which descriptor table the selector refers. A zero indicates the GDT; a one indicates the current LDT.

Requested Privilege Level: Used by the protection mechanism. (Refer to Chapter 6.)

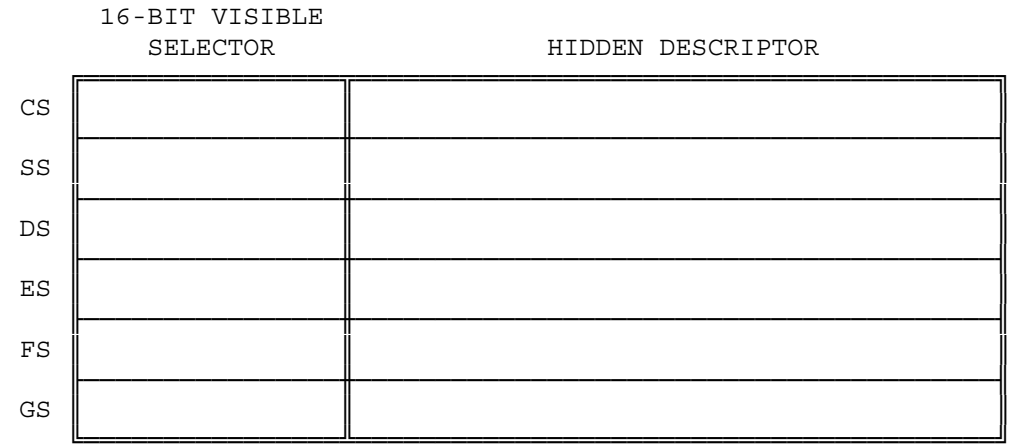
Because the first entry of the GDT is not used by the processor, a selector that has an index of zero and a table indicator of zero (i.e., a selector that points to the first entry of the GDT), can be used as a null selector. The processor does not cause an exception when a segment register (other than CS or SS) is loaded with a null selector. It will, however, cause an exception when the segment register is used to access memory. This feature is useful for initializing unused segment registers so as to trap accidental references.

**Figure 5-6. Format of a Selector**




TI - TABLE INDICATOR
RPL - REQUESTOR'S PRIVILEGE LEVEL

**Figure 5-7. Segment Registers**




## 5.1.4 Segment Registers

The 80386 stores information from descriptors in segment registers, thereby avoiding the need to consult a descriptor table every time it accesses memory.

Every segment register has a "visible" portion and an "invisible" portion, as Figure 5-7 illustrates. The visible portions of these segment address registers are manipulated by programs as if they were simply 16-bit registers. The invisible portions are manipulated by the processor.

The operations that load these registers are normal program instructions (previously described in Chapter 3). These instructions are of two classes:

1. Direct load instructions; for example, MOV, POP, LDS, LSS, LGS, LFS. These instructions explicitly reference the segment registers.

2. Implied load instructions; for example, far CALL and JMP. These instructions implicitly reference the CS register, and load it with a new value.

Using these instructions, a program loads the visible part of the segment register with a 16-bit selector. The processor automatically fetches the base address, limit, type, and other information from a descriptor table and loads them into the invisible part of the segment register.

Because most instructions refer to data in segments whose selectors have already been loaded into segment registers, the processor can add the segment-relative offset supplied by the instruction to the segment base address with no additional overhead.

## 5.2 Page Translation

In the second phase of address transformation, the 80386 transforms a linear address into a physical address. This phase of address transformation implements the basic features needed for page-oriented virtual-memory systems and page-level protection.

The page-translation step is optional. Page translation is in effect only when the PG bit of CR0 is set. This bit is typically set by the operating system during software initialization. The PG bit must be set if the operating system is to implement multiple virtual 8086 tasks, page-oriented protection, or page-oriented virtual memory.

### 5.2.1 Page Frame

A page frame is a 4K-byte unit of contiguous addresses of physical memory. Pages begin onbyte boundaries and are fixed in size.

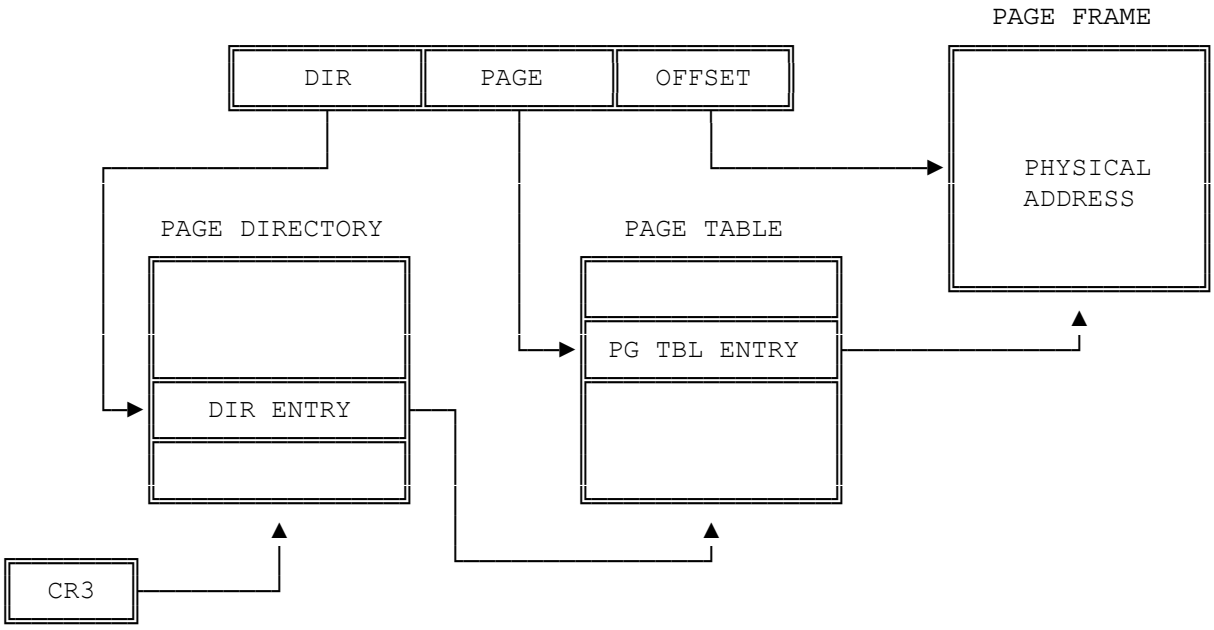
### 5.2.2 Linear Address

A linear address refers indirectly to a physical address by specifying a page table, a page within that table, and an offset within that page. Figure 5-8 shows the format of a linear address.

Figure 5-9 shows how the processor converts the DIR, PAGE, and OFFSET fields of a linear address into the physical address by consulting two levels of page tables. The addressing mechanism uses the DIR field as an index into a page directory, uses the PAGE field as an index into the page table determined by the page directory, and uses the OFFSET field to address a byte within the page determined by the page table.

**Figure 5-8. Format of a Linear Address**

**Figure 5-9. Page Translation**




### 5.2.3 Page Tables

A page table is simply an array of 32-bit page specifiers. A page table is itself a page, and therefore contains 4 Kilobytes of memory or at most 1K 32-bit entries.

Two levels of tables are used to address a page of memory. At the higher level is a page directory. The page directory addresses up to 1K page tables of the second level. A page table of the second level addresses up to 1K pages. All the tables addressed by one page directory, therefore, can address 1M pages ($2^{20}$). Because each page contains 4K bytes $2^{12}$ bytes), the tables of one page directory can span the entire physical address space of the 80386 ($2^{20}$ times $2^{12}$ = $2^{32}$).

The physical address of the current page directory is stored in the CPU register CR3, also called the page directory base register (PDBR). Memory management software has the option of using one page directory for all tasks, one page directory for each task, or some combination of the two. Refer to Chapter 10 for information on initialization of CR3. Refer to Chapter 7 to see how CR3 can change for each task.

### 5.2.4 Page-Table Entries

Entries in either level of page tables have the same format. Figure 5-10 illustrates this format.

### 5.2.4.1 Page Frame Address

The page frame address specifies the physical starting address of a page. Because pages are located on 4K boundaries, the low-order 12 bits are always zero. In a page directory, the page frame address is the address of a page table. In a second-level page table, the page frame address is the address of the page frame that contains the desired memory operand.

### 5.2.4.2 Present Bit

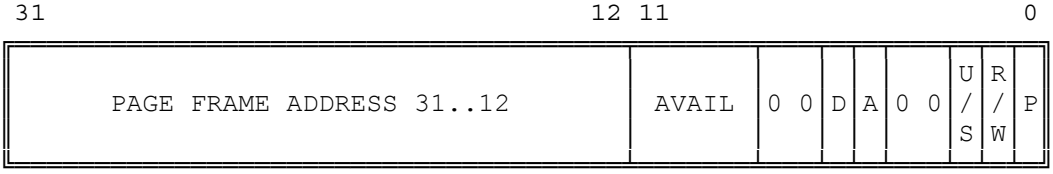
The Present bit indicates whether a page table entry can be used in address translation. P=1 indicates that the entry can be used.

When P=0 in either level of page tables, the entry is not valid for address translation, and the rest of the entry is available for software use; none of the other bits in the entry is tested by the hardware. Figure 5-11 illustrates the format of a page-table entry when P=0.

If P=0 in either level of page tables when an attempt is made to use a page-table entry for address translation, the processor signals a page exception. In software systems that support paged virtual memory, the page-not-present exception handler can bring the required page into physical memory. The instruction that caused the exception can then be reexecuted. Refer to Chapter 9 for more information on exception handlers.

Note that there is no present bit for the page directory itself. The page directory may be not-present while the associated task is suspended, but the operating system must ensure that the page directory indicated by the CR3 image in the TSS is present in physical memory before the task is dispatched. Refer to Chapter 7 for an explanation of the TSS and task dispatching.

**Figure 5-10. Format of a Page Table Entry**




P - PRESENT
R/W - READ/WRITE
U/S - USER/SUPERVISOR
D - DIRTY
AVAIL - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.

**Figure 5-11. Invalid Page Table Entry**


#### 5.2.4.3 Accessed and Dirty Bits

These bits provide data about page usage in both levels of the page tables. With the exception of the dirty bit in a page directory entry, these bits are set by the hardware; however, the processor does not clear any of these bits.

The processor sets the corresponding accessed bits in both levels of page tables to one before a read or write operation to a page.

The processor sets the dirty bit in the second-level page table to one before a write to an address covered by that page table entry. The dirty bit in directory entries is undefined.

An operating system that supports paged virtual memory can use these bits to determine what pages to eliminate from physical memory when the demand for memory exceeds the physical memory available. The operating system is responsible for testing and clearing these bits.

Refer to Chapter 11 for how the 80386 coordinates updates to the accessed and dirty bits in multiprocessor systems.

#### 5.2.4.4 Read/Write and User/Supervisor Bits

These bits are not used for address translation, but are used for page-level protection, which the processor performs at the same time as address translation. Refer to Chapter 6 where protection is discussed in detail.

### 5.2.5 Page Translation Cache

For greatest efficiency in address translation, the processor stores the most recently used page-table data in an on-chip cache. Only if the necessary paging information is not in the cache must both levels of page tables be referenced.

The existence of the page-translation cache is invisible to applications programmers but not to systems programmers; operating-system programmers must flush the cache whenever the page tables are changed. The page-translation cache can be flushed by either of two methods:

1. By reloading CR3 with a MOV instruction; for example:

   MOV CR3, EAX

2. By performing a task switch to a TSS that has a different CR3 image than the current TSS. (Refer to Chapter 7 for more information on task switching.)

## 5.3 Combining Segment and Page Translation

Figure 5-12 combines Figure 5-2 and Figure 5-9 to summarize both phases of the transformation from a logical address to a physical address when paging is enabled. By appropriate choice of options and parameters to both phases, memory-management software can implement several different styles of memory management.

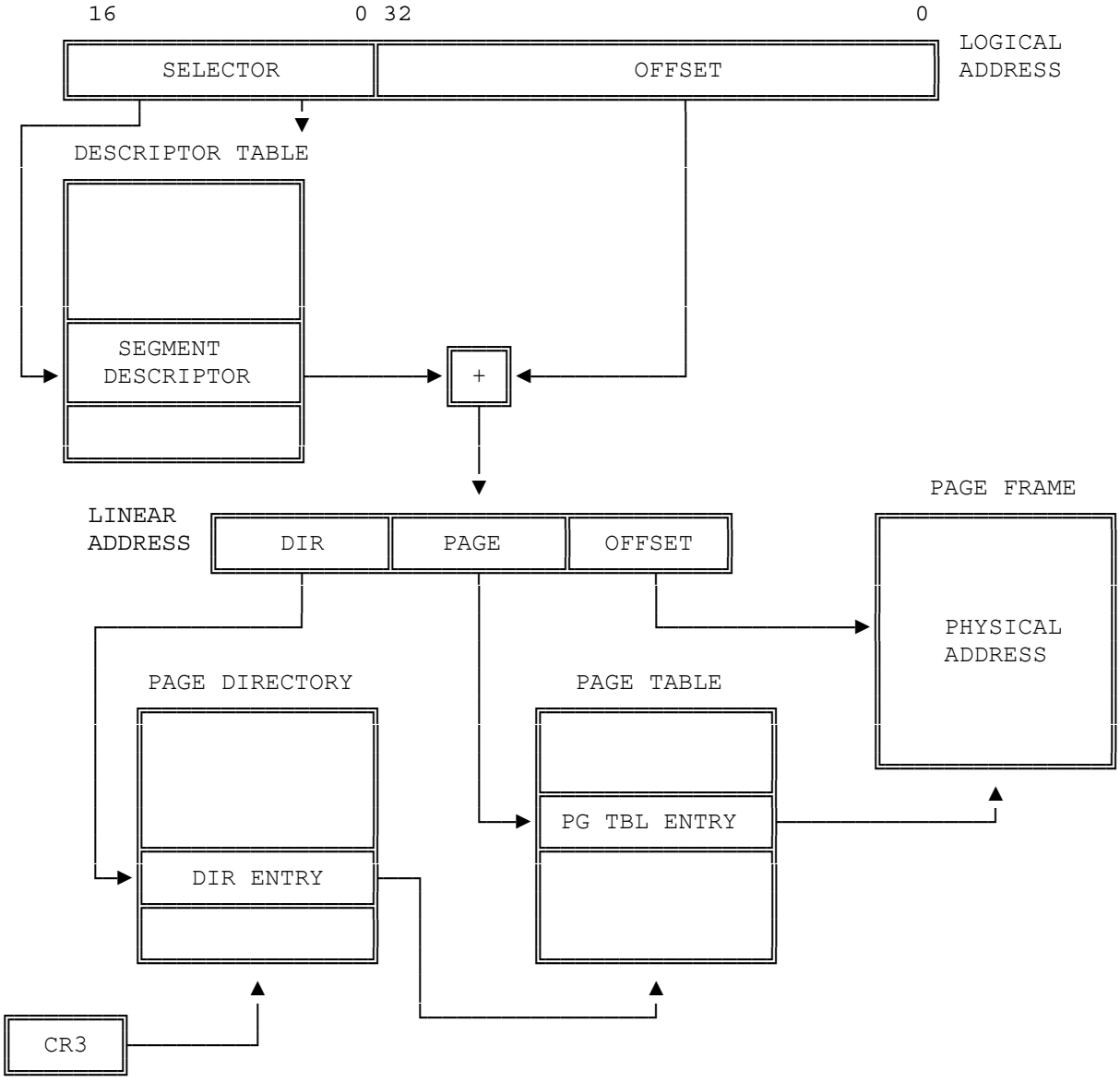
### 5.3.1 "Flat" Architecture

When the 80386 is used to execute software designed for architectures that don't have segments, it may be expedient to effectively "turn off" the segmentation features of the 80386. The 80386 does not have a mode that disables segmentation, but the same effect can be achieved by initially loading the segment registers with selectors for descriptors that encompass the entire 32-bit linear address space. Once loaded, the segment registers don't need to be changed. The 32-bit offsets used by 80386 instructions are adequate to address the entire linear-address space.

### 5.3.2 Segments Spanning Several Pages

The architecture of the 80386 permits segments to be larger or smaller than the size of a page (4 Kilobytes). For example, suppose a segment is used to address and protect a large data structure that spans 132 Kilobytes. In a software system that supports paged virtual memory, it is not necessary for the entire structure to be in physical memory at once. The structure is divided into 33 pages, any number of which may not be present. The applications programmer does not need to be aware that the virtual memory subsystem is paging the structure in this manner.

**Figure 5-12. 80306 Addressing Machanism**




### 5.3.3 Pages Spanning Several Segments

On the other hand, segments may be smaller than the size of a page. For example, consider a small data structure such as a semaphore. Because of the protection and sharing provided by segments (refer to Chapter 6), it may be useful to create a separate segment for each semaphore. But, because a system may need many semaphores, it is not efficient to allocate a page for each. Therefore, it may be useful to cluster many related segments within a page.

### 5.3.4 Non-Aligned Page and Segment Boundaries

The architecture of the 80386 does not enforce any correspondence between the boundaries of pages and segments. It is perfectly permissible for a page to contain the end of one segment and the beginning of another. Likewise, a segment may contain the end of one page and the beginning of another.

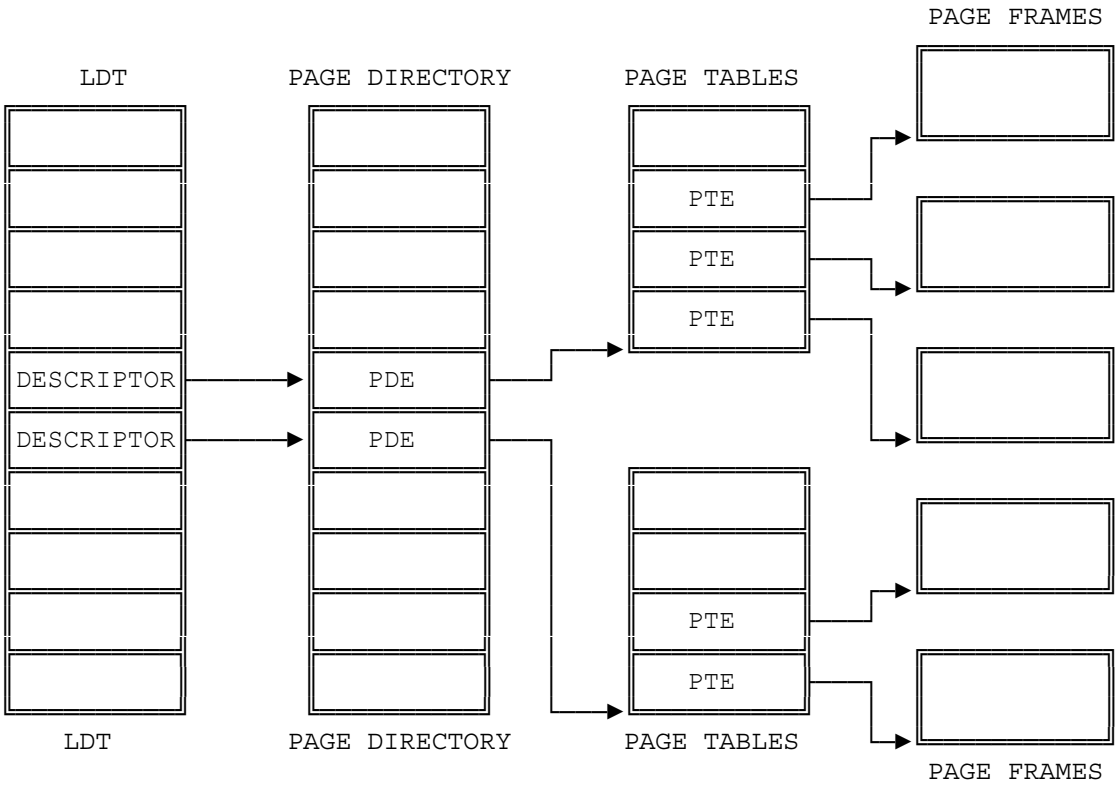
### 5.3.5 Aligned Page and Segment Boundaries

Memory-management software may be simpler, however, if it enforces some correspondence between page and segment boundaries. For example, if segments are allocated only in units of one page, the logic for segment and page allocation can be combined. There is no need for logic to account for partially used pages.

### 5.3.6 Page-Table per Segment

An approach to space management that provides even further simplification of space-management software is to maintain a one-to-one correspondence between segment descriptors and page-directory entries, as Figure 5-13 illustrates. Each descriptor has a base address in which the low-order 22 bits are zero; in other words, the base address is mapped by the first entry of a page table. A segment may have any limit from 1 to 4 megabytes. Depending on the limit, the segment is contained in from 1 to 1K page frames. A task is thus limited to 1K segments (a sufficient number for many applications), each containing up to 4 Mbytes. The descriptor, the corresponding page-directory entry, and the corresponding page table can be allocated and deallocated simultaneously.

**Figure 5-13. Descriptor per Page Table**



# Chapter 6 Protection

## 6.1 Why Protection?

The purpose of the protection features of the 80386 is to help detect and identify bugs. The 80386 supports sophisticated applications that may consist of hundreds or thousands of program modules. In such applications, the question is how bugs can be found and eliminated as quickly as possible and how their damage can be tightly confined. To help debug applications faster and make them more robust in production, the 80386 contains mechanisms to verify memory accesses and instruction execution for conformance to protection criteria. These mechanisms may be used or ignored, according to system design objectives.

## 6.2 Overview of 80386 Protection Mechanisms

Protection in the 80386 has five aspects:

1. Type checking
2. Limit checking
3. Restriction of addressable domain
4. Restriction of procedure entry points
5. Restriction of instruction set

The protection hardware of the 80386 is an integral part of the memory management hardware. Protection applies both to segment translation and to page translation.

Each reference to memory is checked by the hardware to verify that it satisfies the protection criteria. All these checks are made before the memory cycle is started; any violation prevents that cycle from starting and results in an exception. Since the checks are performed concurrently with address formation, there is no performance penalty.

Invalid attempts to access memory result in an exception. Refer to Chapter 9 for an explanation of the exception mechanism. The present chapter defines the protection violations that lead to exceptions.

The concept of "privilege" is central to several aspects of protection (numbers 3, 4, and 5 in the preceeding list). Applied to procedures, privilege is the degree to which the procedure can be trusted not to make a mistake that might affect other procedures or data. Applied to data, privilege is the degree of protection that a data structure should have from less trusted procedures.

The concept of privilege applies both to segment protection and to page protection.

## 6.3 Segment-Level Protection

All five aspects of protection apply to segment translation:

1. Type checking
2. Limit checking
3. Restriction of addressable domain
4. Restriction of procedure entry points
5. Restriction of instruction set

The segment is the unit of protection, and segment descriptors store protection parameters. Protection checks are performed automatically by the CPU when the selector of a segment descriptor is loaded into a segment register and with every segment access. Segment registers hold the protection parameters of the currently addressable segments.

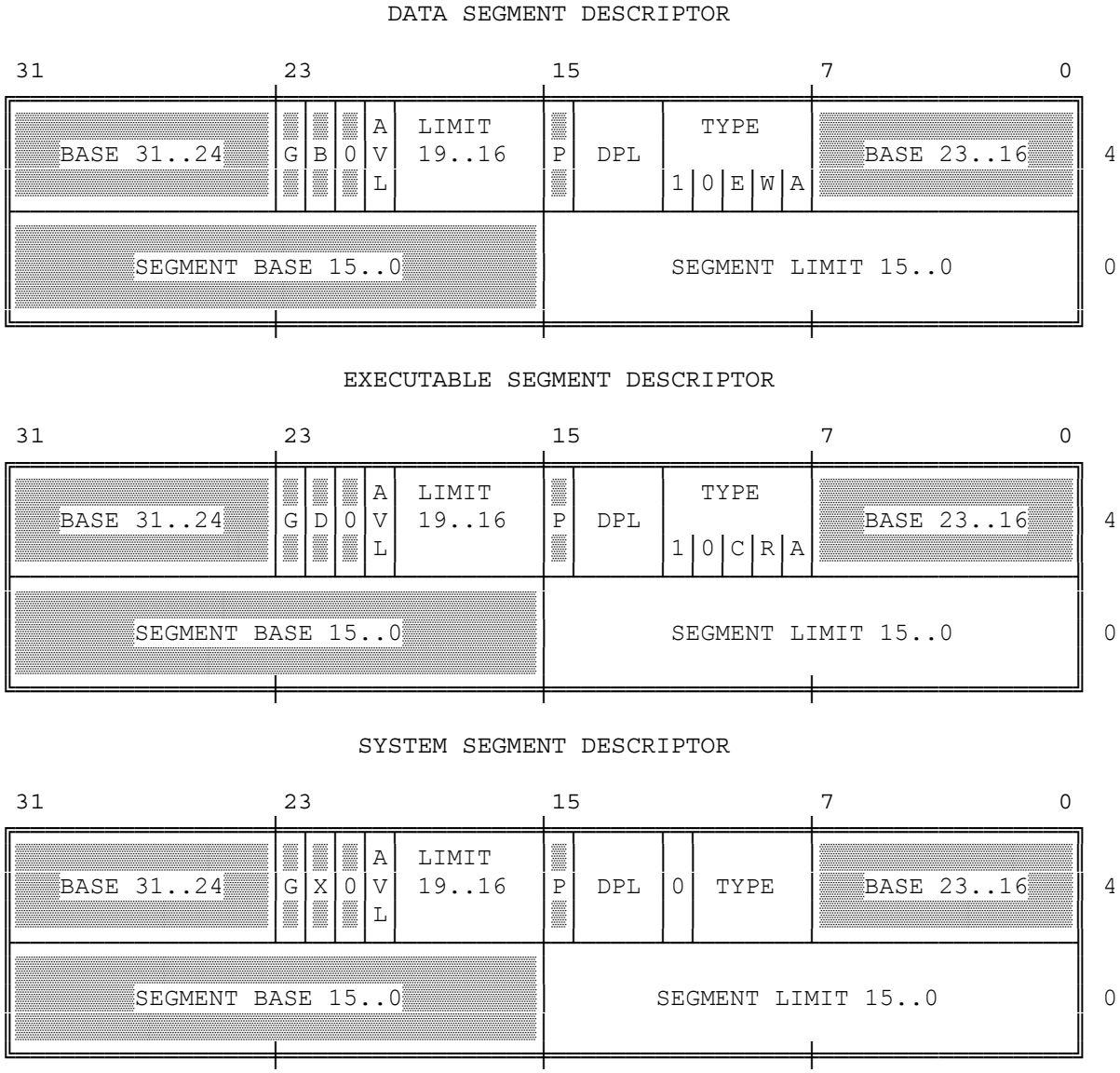
### 6.3.1 Descriptors Store Protection Parameters

Figure 6-1 highlights the protection-related fields of segment descriptors.

The protection parameters are placed in the descriptor by systems software at the time a descriptor is created. In general, applications programmers do not need to be concerned about protection parameters.

When a program loads a selector into a segment register, the processor loads not only the base address of the segment but also protection information. Each segment register has bits in the invisible portion for storing base, limit, type, and privilege level; therefore, subsequent protection checks on the same segment do not consume additional clock cycles.

**Figure 6-1. Protection Fields of Segment Descriptors**




A - ACCESSED
AVL - AVAILABLE FOR PROGRAMMERS USE
B - BIG
C - CONFORMING
D - DEFAULT
DPL - DESCRIPTOR PRIVILEGE LEVEL
E - EXPAND-DOWN
G - GRANULARITY
P - SEGMENT PRESENT
R - READABLE
W - WRITABLE

#### 6.3.1.1 Type Checking

The TYPE field of a descriptor has two functions:

1. It distinguishes among different descriptor formats.
2. It specifies the intended usage of a segment.

Besides the descriptors for data and executable segments commonly used by applications programs, the 80386 has descriptors for special segments used by the operating system and for gates. Table 6-1 lists all the types defined for system segments and gates. Note that not all descriptors define segments; gate descriptors have a different purpose that is discussed later in this chapter.

The type fields of data and executable segment descriptors include bits which further define the purpose of the segment (refer to Figure 6-1):

- The writable bit in a data-segment descriptor specifies whether instructions can write into the segment.

- The readable bit in an executable-segment descriptor specifies whether instructions are allowed to read from the segment (for example, to access constants that are stored with instructions). A readable, executable segment may be read in two ways:

1. Via the CS register, by using a CS override prefix.

2. By loading a selector of the descriptor into a data-segment register (DS, ES, FS,or GS).

Type checking can be used to detect programming errors that would attempt to use segments in ways not intended by the programmer. The processor examines type information on two kinds of occasions:

1. When a selector of a descriptor is loaded into a segment register. Certain segment registers can contain only certain descriptor types; for example:

- The CS register can be loaded only with a selector of an executable segment.

- Selectors of executable segments that are not readable cannot be loaded into data-segment registers.

- Only selectors of writable data segments can be loaded into SS.

2. When an instruction refers (implicitly or explicitly) to a segment register. Certain segments can be used by instructions only in certain predefined ways; for example:

- No instruction may write into an executable segment.

- No instruction may write into a data segment if the writable bit is not set.

- No instruction may read an executable segment unless the readable bit is set.

**Table 6-1. System and Gate Descriptor Types**

| Code | Type of Segment or Gate |
|---|---|
| 0 | -reserved |
| 1 | Available 286 TSS |
| 2 | LDT |
| 3 | Busy 286 TSS |
| 4 | Call Gate |
| 5 | Task Gate |
| 6 | 286 Interrupt Gate |
| 7 | 286 Trap Gate |
| 8 | -reserved |
| 9 | Available 386 TSS |
| A | -reserved |
| B | Busy 386 TSS |
| C | 386 Call Gate |
| D | -reserved |
| E | 386 Interrupt Gate |
| F | 386 Trap Gate |

### 6.3.1.2 Limit Checking

The limit field of a segment descriptor is used by the processor to prevent programs from addressing outside the segment. The processor's interpretation of the limit depends on the setting of the G (granularity) bit. For data segments, the processor's interpretation of the limit depends also on the E-bit (expansion-direction bit) and the B-bit (big bit) (refer to Table 6-2).

When G=0, the actual limit is the value of the 20-bit limit field as it appears in the descriptor. In this case, the limit may range from 0 to 0FFFFFH ($2^{20}$-1 or 1 megabyte). When G=1, the processor appends 12 low-order one-bits to the value in the limit field. In this case the actual limit may range from 0FFFH ($2^{12}$-1 or 4 kilobytes) to 0FFFFFFFFH($2^{32}$-1 or 4 gigabytes).

For all types of segments except expand-down data segments, the value of the limit is one less than the size (expressed in bytes) of the segment. The processor causes a general-protection exception in any of these cases:

- Attempt to access a memory byte at an address > limit.
- Attempt to access a memory word at an address ≥limit.
- Attempt to access a memory doubleword at an address ≥(limit-2).

For expand-down data segments, the limit has the same function but is interpreted differently. In these cases the range of valid addresses is from limit + 1 to either 64K or $2^{32}$-1 (4 Gbytes) depending on the B-bit. An expand-down segment has maximum size when the limit is zero.

The expand-down feature makes it possible to expand the size of a stack by copying it to a larger segment without needing also to update intrastack pointers.

The limit field of descriptors for descriptor tables is used by the processor to prevent programs from selecting a table entry outside the descriptor table. The limit of a descriptor table identifies the last valid byte of the last descriptor in the table. Since each descriptor is eight bytes long, the limit value is N * 8 - 1 for a table that can contain up to N descriptors.

Limit checking catches programming errors such as runaway subscripts and invalid pointer calculations. Such errors are detected when they occur, so that identification of the cause is easier. Without limit checking, such errors could corrupt other modules; the existence of such errors would not be discovered until later, when the corrupted module behaves incorrectly, and when identification of the cause is difficult.

**Table 6-2. Useful Combinations of E, G, and B Bits**

| Case: | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Expansion Direction | U | U | D | D |
| G-bit | 0 | 1 | 0 | 1 |
| B-bit | X | X | 0 | 1 |
| Lower bound is: | | | | |
| 0 | X | X | | |
| LIMIT+1 | | | X | |
| shl(LIMIT,12,1)+1 | | | | X |
| Upper bound is: | | | | |
| LIMIT | X | | | |
| shl(LIMIT,12,1) | | X | | |
| 64K-1 | | | X | |
| 4G-1 | | | | X |
| Max seg size is: | | | | |
| 64K | X | | | |
| 64K-1 | | X | | |
| 4G-4K | | | X | |
| 4G | | | | X |
| Min seg size is: | | | | |
| 0 | X | X | | |
| 4K | | | X | X |

shl (X, 12, 1) = shift X left by 12 bits inserting one-bits on the right

#### 6.3.1.3 Privilege Levels

The concept of privilege is implemented by assigning a value from zero to three to key objects recognized by the processor. This value is called the privilege level. The value zero represents the greatest privilege, the value three represents the least privilege. The following processor-recognized objects contain privilege levels:

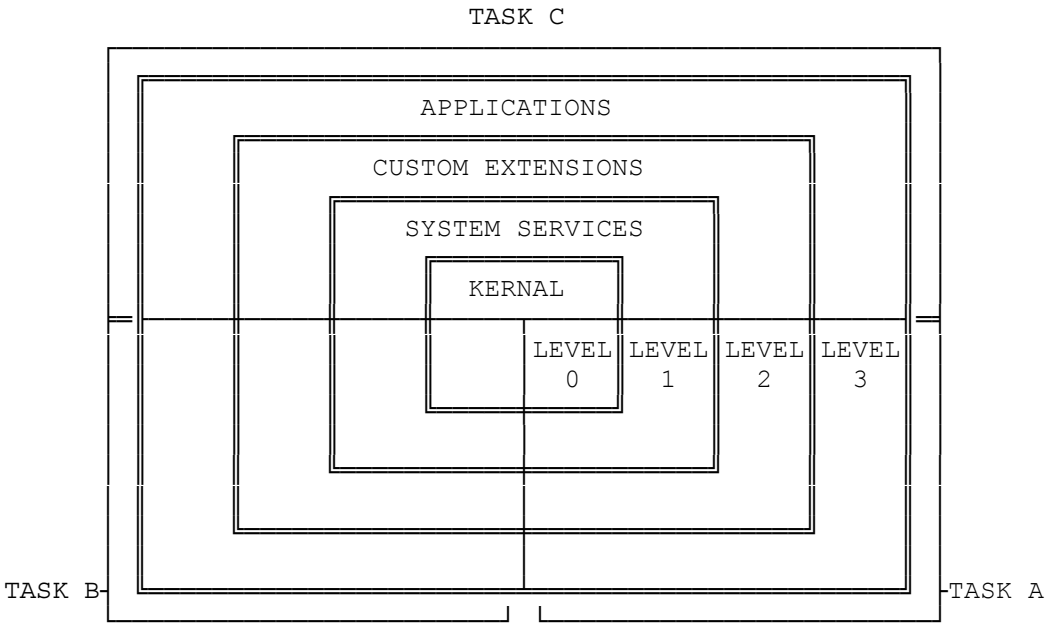
- Descriptors contain a field called the descriptor privilege level (DPL).
- Selectors contain a field called the requestor's privilege level (RPL). The RPL is intended to represent the privilege level of the procedure that originates a selector.
- An internal processor register records the current privilege level (CPL). Normally the CPL is equal to the DPL of the segment that the processor is currently executing. CPL changes as control is transferred to segments with differing DPLs.

The processor automatically evaluates the right of a procedure to access another segment by comparing the CPL to one or more other privilege levels. The evaluation is performed at the time the selector of a descriptor is loaded into a segment register. The criteria used for evaluating access to data differs from that for evaluating transfers of control to executable segments; therefore, the two types of access are considered separately in the following sections.

Figure 6-2 shows how these levels of privilege can be interpreted as rings of protection. The center is for the segments containing the most critical software, usually the kernel of the operating system. Outer rings are for the segments of less critical software.

It is not necessary to use all four privilege levels. Existing software that was designed to use only one or two levels of privilege can simply ignore the other levels offered by the 80386. A one-level system should use privilege level zero; a two-level system should use privilege levels zero and three.

**Figure 6-2. Levels of Privilege**




## 6.3.2 Restricting Access to Data

To address operands in memory, an 80386 program must load the selector of a data segment into a data-segment register (DS, ES, FS, GS, SS). The processor automatically evaluates access to a data segment by comparing privilege levels. The evaluation is performed at the time a selector for the descriptor of the target segment is loaded into the data-segment register. As Figure 6-3 shows, three different privilege levels enter into this type of privilege check:

1. The CPL (current privilege level).

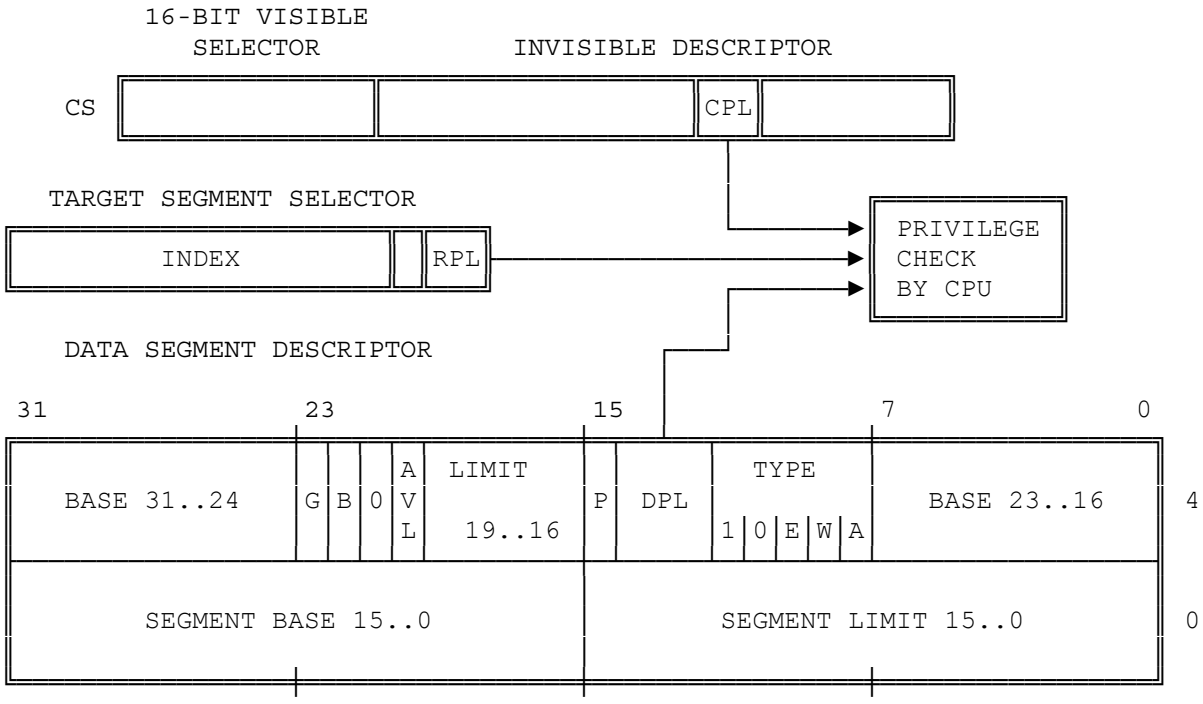
2. The RPL (requestor's privilege level) of the selector used to specify the target segment.

3. The DPL of the descriptor of the target segment.

Instructions may load a data-segment register (and subsequently use the target segment) only if the DPL of the target segment is numerically greater than or equal to the maximum of the CPL and the selector's RPL. In other words, a procedure can only access data that is at the same or less privileged level.

The addressable domain of a task varies as CPL changes. When CPL is zero, data segments at all privilege levels are accessible; when CPL is one, only data segments at privilege levels one through three are accessible; when CPL is three, only data segments at privilege level three are accessible. This property of the 80386 can be used, for example, to prevent applications procedures from reading or changing tables of the operating system.

**Figure 6-3. Privilege Check for Data Access**




CPL - CURRENT PRIVILEGE LEVEL
RPL - REQUESTOR'S PRIVILEGE LEVEL
DPL - DESCRIPTOR PRIVILEGE LEVEL

### 6.3.2.1 Accessing Data in Code Segments

Less common than the use of data segments is the use of code segments to store data. Code segments may legitimately hold constants; it is not possible to write to a segment described as a code segment. The following methods of accessing data in code segments are possible:

1. Load a data-segment register with a selector of a nonconforming, readable, executable segment.

2. Load a data-segment register with a selector of a conforming, readable, executable segment.

3. Use a CS override prefix to read a readable, executable segment whose selector is already loaded in the CS register.

The same rules as for access to data segments apply to case 1. Case 2 is always valid because the privilege level of a segment whose conforming bit is set is effectively the same as CPL regardless of its DPL. Case 3 always valid because the DPL of the code segment in CS is, by definition, equal to CPL.

### 6.3.3 Restricting Control Transfers

With the 80386, control transfers are accomplished by the instructions JMP, CALL, RET, INT, and IRET, as well as by the exception and interrupt mechanisms. Exceptions and interrupts are special cases that Chapter 9 covers. This chapter discusses only JMP, CALL, and RET instructions.

The "near" forms of JMP, CALL, and RET transfer within the current code segment, and therefore are subject only to limit checking. The processor ensures that the destination of the JMP, CALL, or RET instruction does not exceed the limit of the current executable segment. This limit is cached in the CS register; therefore, protection checks for near transfers require no extra clock cycles.

The operands of the "far" forms of JMP and CALL refer to other segments; therefore, the processor performs privilege checking. There are two ways a JMP or CALL can refer to another segment:

1. The operand selects the descriptor of another executable segment.

2. The operand selects a call gate descriptor. This gated form of transfer is discussed in a later section on call gates.

As Figure 6-4 shows, two different privilege levels enter into a privilege check for a control transfer that does not use a call gate:

1. The CPL (current privilege level).
2. The DPL of the descriptor of the target segment.

Normally the CPL is equal to the DPL of the segment that the processor is currently executing. CPL may, however, be greater than DPL if the conforming bit is set in the descriptor of the current executable segment. The processor keeps a record of the CPL cached in the CS register; this value can be different from the DPL in the descriptor of the code segment.

The processor permits a JMP or CALL directly to another segment only if one of the following privilege rules is satisfied:

- DPL of the target is equal to CPL.

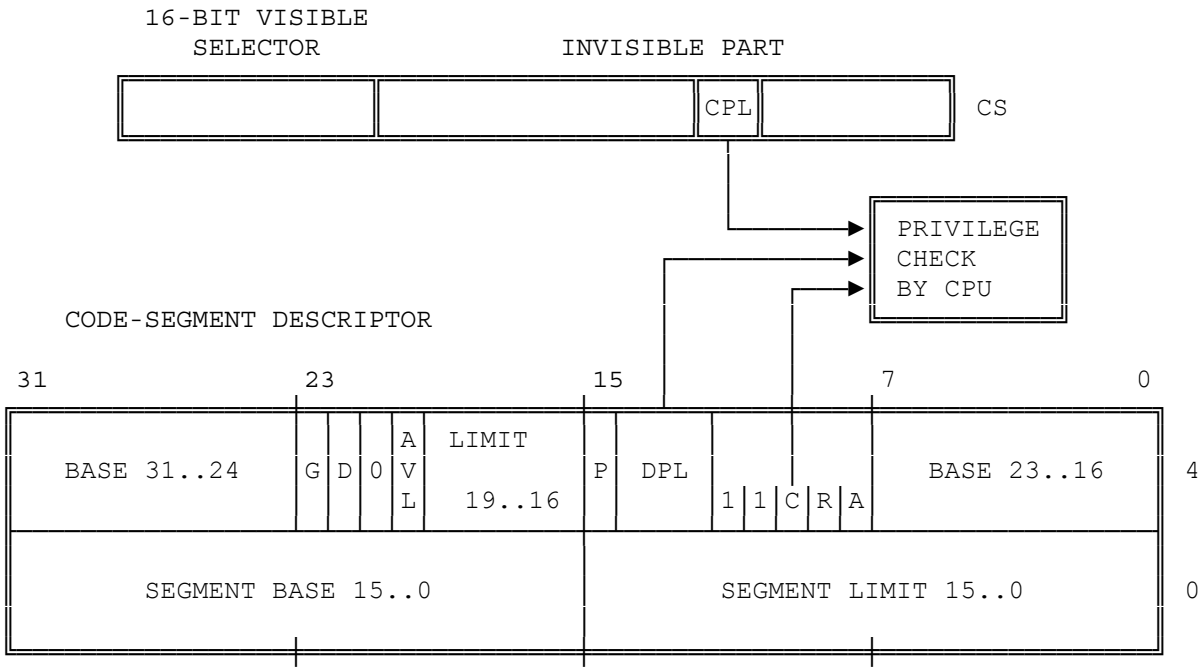
- The conforming bit of the target code-segment descriptor is set, and the DPL of the target is less than or equal to CPL.

An executable segment whose descriptor has the conforming bit set is called a conforming segment. The conforming-segment mechanism permits sharing of procedures that may be called from various privilege levels but should execute at the privilege level of the calling procedure. Examples of such procedures include math libraries and some exception handlers. When control is transferred to a conforming segment, the CPL does not change. This is the only case when CPL may be unequal to the DPL of the current executable segment.

Most code segments are not conforming. The basic rules of privilege above mean that, for nonconforming segments, control can be transferred without a gate only to executable segments at the same level of privilege. There is a need, however, to transfer control to (numerically) smaller privilege

levels; this need is met by the CALL instruction when used with call-gate descriptors, which are explained in the next section. The JMP instruction may never transfer control to a nonconforming segment whose DPL does not equal CPL.

**Figure 6-4. Privilege Check for Control Transfer without Gate**




CPL - CURRENT PRIVILEGE LEVEL
DPL - DESCRIPTOR PRIVILEGE LEVEL
C - CONFORMING BIT

## 6.3.4 Gate Descriptors Guard Procedure Entry Points

To provide protection for control transfers among executable segments at different privilege levels, the 80386 uses gate descriptors. There are four kinds of gate descriptors:

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

This chapter is concerned only with call gates. Task gates are used for task switching, and therefore are discussed in Chapter 7. Chapter 9 explains how trap gates and interrupt gates are used by exceptions and interrupts. Figure 6-5 illustrates the format of a call gate. A call gate descriptor may reside in the GDT or in an LDT, but not in the IDT.

A call gate has two primary functions:

1. To define an entry point of a procedure.
2. To specify the privilege level of the entry point.

Call gate descriptors are used by call and jump instructions in the same manner as code segment descriptors. When the hardware recognizes that the destination selector refers to a gate descriptor, the operation of the instruction is expanded as determined by the contents of the call gate.

The selector and offset fields of a gate form a pointer to the entry point of a procedure. A call gate guarantees that all transitions to another segment go to a valid entry point, rather than possibly into the middle of a procedure (or worse, into the middle of an instruction). The far pointer operand of the control transfer instruction does not point to the segment and offset of the target instruction; rather, the selector part of the pointer selects a gate, and the offset is not used. Figure 6-6 illustrates this style of addressing.

As Figure 6-7 shows, four different privilege levels are used to check the validity of a control transfer via a call gate:

1. The CPL (current privilege level).

2. The RPL (requestor's privilege level) of the selector used to specify the call gate.

3. The DPL of the gate descriptor.

4. The DPL of the descriptor of the target executable segment.

The DPL field of the gate descriptor determines what privilege levels can use the gate. One code segment can have several procedures that are intended for use by different privilege levels. For example, an operating system may have some services that are intended to be used by applications, whereas others may be intended only for use by other systems software.

Gates can be used for control transfers to numerically smaller privilege levels or to the same privilege level (though they are not necessary for transfers to the same level). Only CALL instructions can use gates to transfer to smaller privilege levels. A gate may be used by a JMP instruction only to transfer to an executable segment with the same privilege level or to a conforming segment.

For a JMP instruction to a nonconforming segment, both of the following privilege rules must be satisfied; otherwise, a general protection exception results.

```
MAX (CPL,RPL) ≤ gate DPL
target segment DPL = CPL
```

For a CALL instruction (or for a JMP instruction to a conforming segment), both of the following privilege rules must be satisfied; otherwise, a general protection exception results.

```
MAX (CPL,RPL) ≤ gate DPL
target segment DPL ≤ CPL
```

**Figure 6-5. Format of 80386 Call Gate**

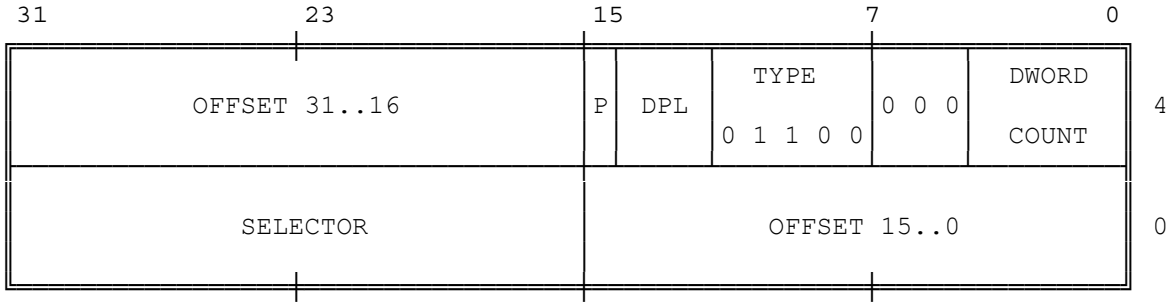



**Figure 6-6. Indirect Transfer via Call Gate**

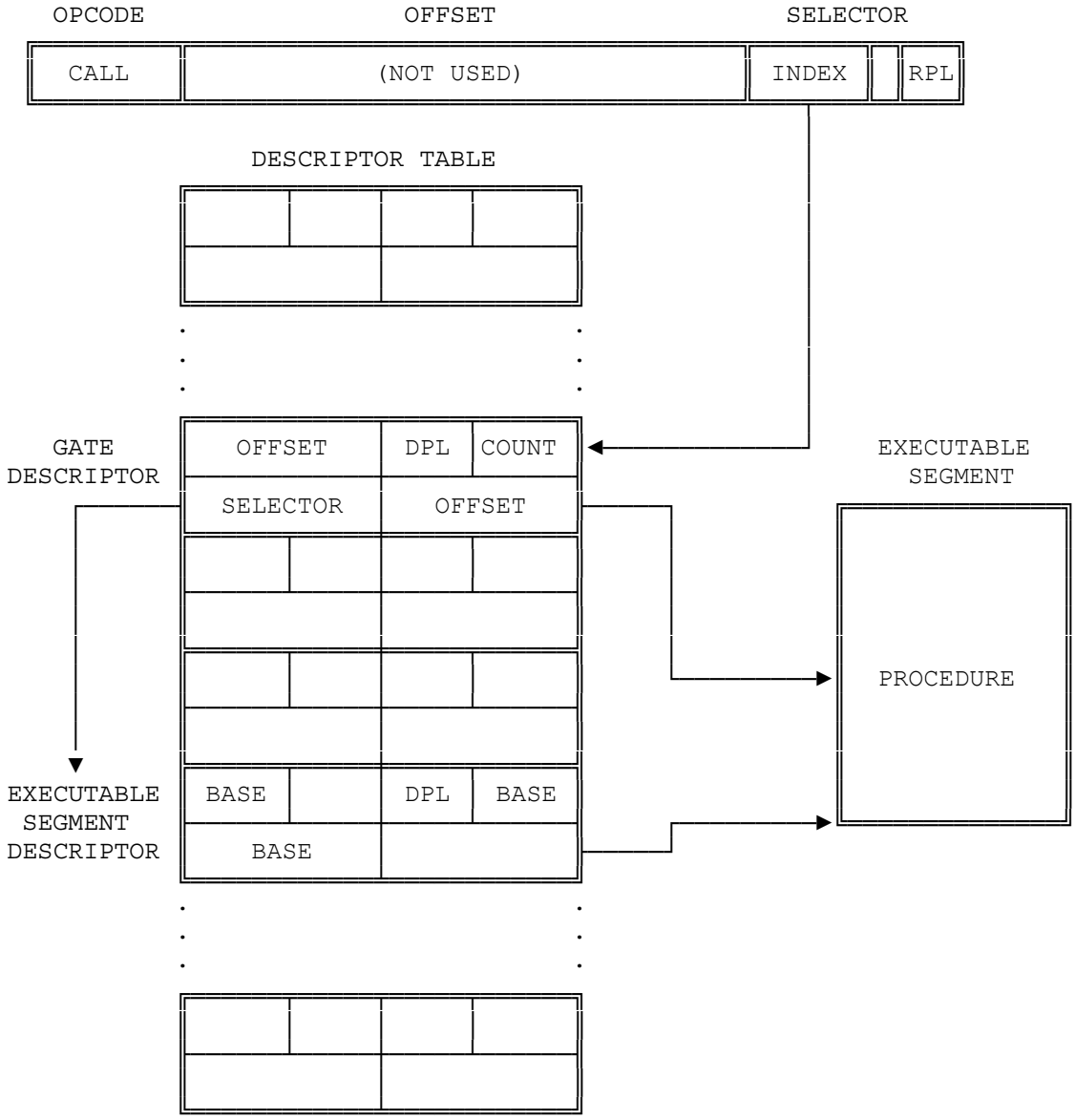
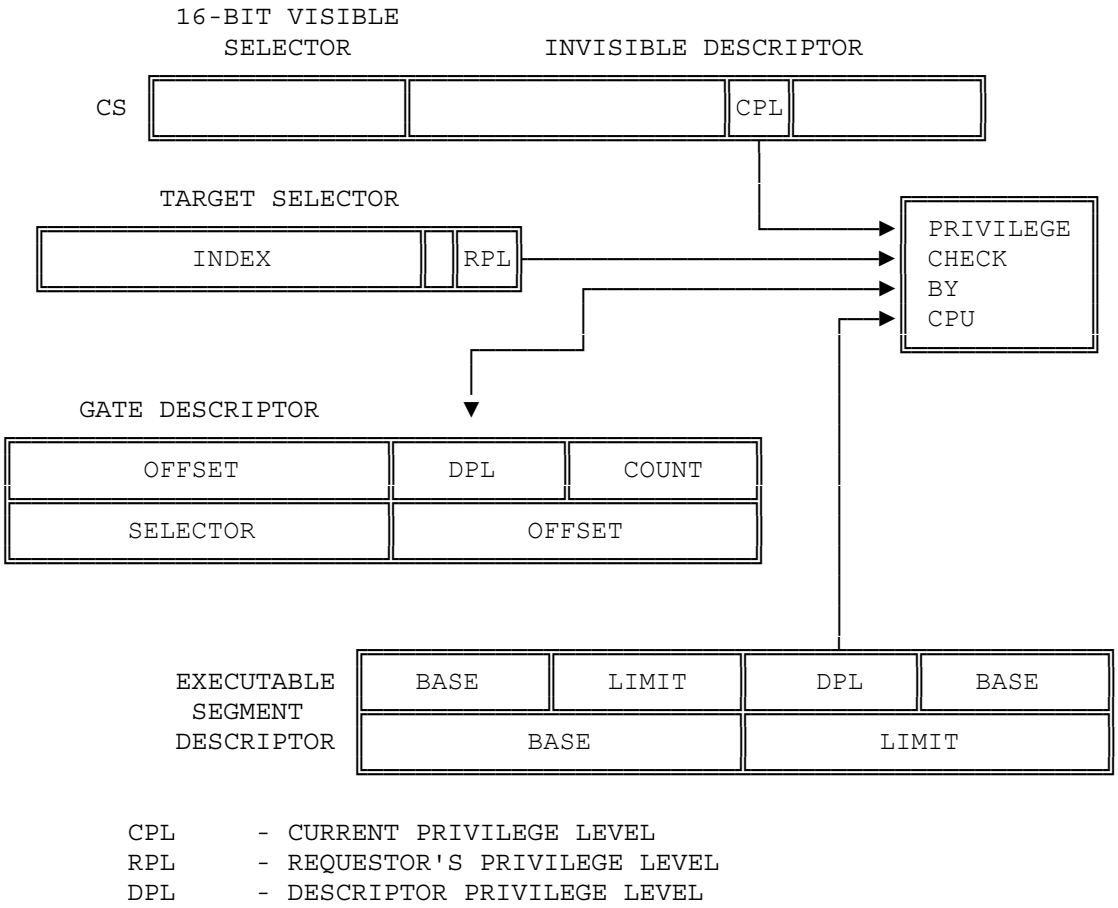


**Figure 6-7. Privilege Check via Call Gate**




CPL - CURRENT PRIVILEGE LEVEL
RPL - REQUESTOR'S PRIVILEGE LEVEL
DPL - DESCRIPTOR PRIVILEGE LEVEL

### 6.3.4.1 Stack Switching

If the destination code segment of the call gate is at a different privilege level than the CPL, an interlevel transfer is being requested.

To maintain system integrity, each privilege level has a separate stack. These stacks assure sufficient stack space to process calls from less privileged levels. Without them, a trusted procedure would not work correctly if the calling procedure did not provide sufficient space on the caller's stack.

The processor locates these stacks via the task state segment (see Figure 6-8). Each task has a separate TSS, thereby permitting tasks to have separate stacks. Systems software is responsible for creating TSSs and placing correct stack pointers in them. The initial stack pointers in the TSS are strictly read-only values. The processor never changes them during the course of execution.

When a call gate is used to change privilege levels, a new stack is selected by loading a pointer value from the Task State Segment (TSS). The processor uses the DPL of the target code segment (the new CPL) to index the initial stack pointer for PL 0, PL 1, or PL 2.

The DPL of the new stack data segment must equal the new CPL; if it does not, a stack exception occurs. It is the responsibility of systems software to create stacks and stack-segment descriptors for all privilege levels that are used. Each stack must contain enough space to hold the old SS:ESP, the return address, and all parameters and local variables that may be required to process a call.

As with intralevel calls, parameters for the subroutine are placed on the stack. To make privilege transitions transparent to the called procedure, the processor copies the parameters to the new stack. The count field of a call gate tells the processor how many doublewords (up to 31) to copy from the caller's stack to the new stack. If the count is zero, no parameters are copied.

The processor performs the following stack-related steps in executing an interlevel CALL.

1. The new stack is checked to assure that it is large enough to hold the parameters and linkages; if it is not, a stack fault occurs with an error code of 0.

2. The old value of the stack registers SS:ESP is pushed onto the new stack as two doublewords.

3. The parameters are copied.

4. A pointer to the instruction after the CALL instruction (the former value of CS:EIP) is pushed onto the new stack. The final value of SS:ESP points to this return pointer on the new stack.

Figure 6-9 illustrates the stack contents after a successful interlevel call.

The TSS does not have a stack pointer for a privilege level 3 stack, because privilege level 3 cannot be called by any procedure at any other privilege level.

Procedures that may be called from another privilege level and that require more than the 31 doublewords for parameters must use the saved SS:ESP link to access all parameters beyond the last doubleword copied.

A call via a call gate does not check the values of the words copied onto the new stack. The called procedure should check each parameter for validity. A later section discusses how the ARPL, VERR, VERW, LSL, and LAR instructions can be used to check pointer values.

**Figure 6-8. Initial Stack Pointers of TSS**

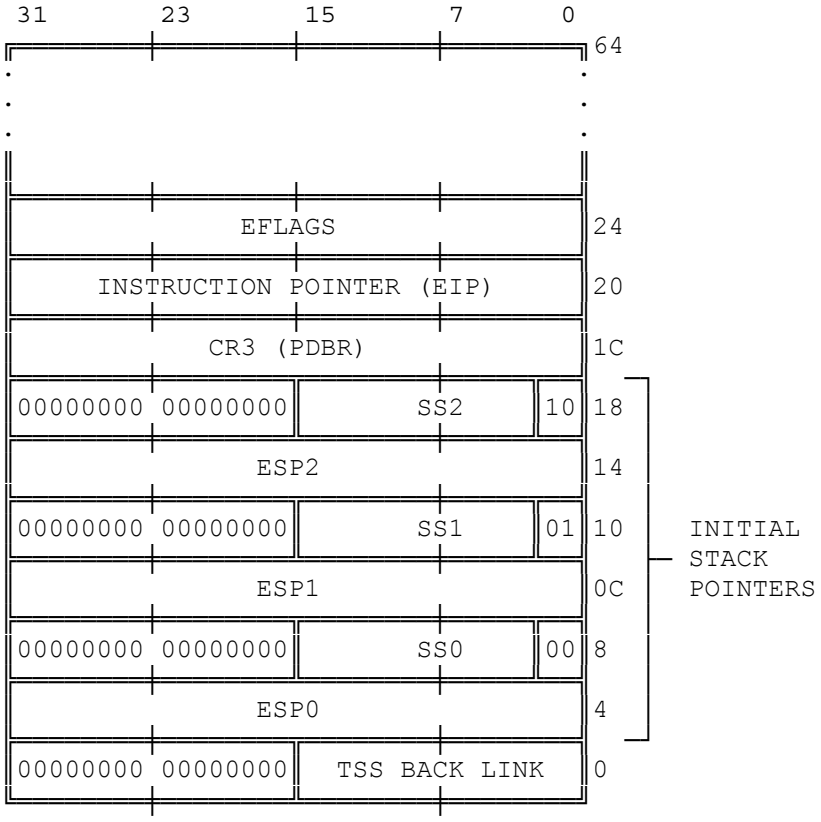
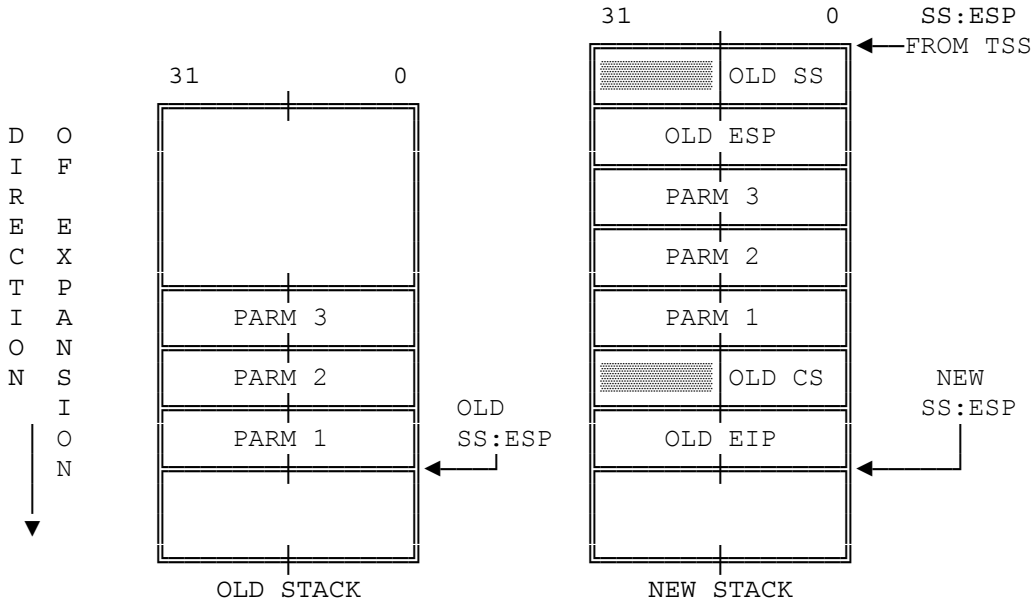



**Figure 6-9. Stack Contents after an Interlevel Call**



#### 6.3.4.2 Returning from a Procedure

The "near" forms of the RET instruction transfer control within the current code segment and therefore are subject only to limit checking. The offset of the instruction following the corresponding CALL, is popped from the stack. The processor ensures that this offset does not exceed the limit of the current executable segment.

The "far" form of the RET instruction pops the return pointer that was pushed onto the stack by a prior far CALL instruction. Under normal conditions, the return pointer is valid, because of its relation to the prior CALL or INT. Nevertheless, the processor performs privilege checking because of the possibility that the current procedure altered the pointer or failed to properly maintain the stack. The RPL of the CS selector popped off the stack by the return instruction identifies the privilege level of the calling procedure.

An intersegment return instruction can change privilege levels, but only toward procedures of lesser privilege. When the RET instruction encounters a saved CS value whose RPL is numerically greater than the CPL, an interlevel return occurs. Such a return follows these steps:

1. The checks shown in Table 6-3 are made, and CS:EIP and SS:ESP are loaded with their former values that were saved on the stack.

2. The old SS:ESP (from the top of the current stack) value is adjusted by the number of bytes indicated in the RET instruction. The resulting ESP value is not compared to the limit of the stack segment. If ESP is beyond the limit, that fact is not recognized until the next stack operation. (The SS:ESP value of the returning procedure is not preserved; normally, this value is the same as that contained in the TSS.)

3. The contents of the DS, ES, FS, and GS segment registers are checked. If any of these registers refer to segments whose DPL is greater than the new CPL (excluding conforming code segments), the segment register is loaded with the null selector (INDEX = 0, TI = 0). The RET instruction itself does not signal exceptions in these cases; however, any subsequent memory reference that attempts to use a segment register that contains the null selector will cause a general protection exception. This prevents less privileged code from accessing more privileged segments using selectors left in the segment registers by the more privileged procedure.

### 6.3.5 Some Instructions are Reserved for Operating System

Instructions that have the power to affect the protection mechanism or to influence general system performance can only be executed by trusted procedures. The 80386 has two classes of such instructions:

1. Privileged instructions —— those used for system control.

2. Sensitive instructions —— those used for I/O and I/O related activities.

**Table 6-3. Interlevel Return Checks**

| Type of Check | Exception | |
|---|---|---|
| SF Stack Fault | | |
| GP General Protection Exception | | |
| NP Segment-Not-Present Exception | Error Code | |
| ESP is within current SS segment | SF | 0 |
| ESP + 7 is within current SS segment | SF | 0 |
| RPL of return CS is greater than CPL | GP | Return CS |
| Return CS selector is not null | GP | Return CS |
| Return CS segment is within descriptor table limit | GP | Return CS |
| Return CS descriptor is a code segment | GP | Return CS |
| Return CS segment is present | NP | Return CS |
| DPL of return nonconforming code segment = RPL of return CS, or DPL of return conforming code segment ≤ RPL of return CS | GP | Return CS |
| ESP + N + 15 is within SS segment | | |
| N Immediate Operand of RET N Instruction | SF | Return SS |
| SS selector at ESP + N + 12 is not null | GP | Return SS |
| SS selector at ESP + N + 12 is within descriptor table limit | GP | Return SS |
| SS descriptor is writable data segment | GP | Return SS |
| SS segment is present | SF | Return SS |
| Saved SS segment DPL = RPL of saved CS | GP | Return SS |
| Saved SS selector RPL = Saved SS segment DPL | GP | Return SS |

### 6.3.5.1 Privileged Instructions

The instructions that affect system data structures can only be executed when CPL is zero. If the CPU encounters one of these instructions when CPL is greater than zero, it signals a general protection exception. These instructions include:

| | |
|---|---|
| CLTS | —— Clear Task—Switched Flag |
| HLT | —— Halt Processor |
| LGDT | —— Load GDL Register |
| LIDT | —— Load IDT Register |
| LLDT | —— Load LDT Register |
| LMSW | —— Load Machine Status Word |
| LTR | —— Load Task Register |
| MOV to/from CRn | —— Move to Control Register n |
| MOV to /from DRn | —— Move to Debug Register n |
| MOV to/from TRn | —— Move to Test Register n |

#### 6.3.5.2 Sensitive Instructions

Instructions that deal with I/O need to be restricted but also need to be executed by procedures executing at privilege levels other than zero. The mechanisms for restriction of I/O operations are covered in detail in Chapter 8, "Input/Output".

### 6.3.6 Instructions for Pointer Validation

Pointer validation is an important part of locating programming errors. Pointer validation is necessary for maintaining isolation between the privilege levels. Pointer validation consists of the following steps:

1. Check if the supplier of the pointer is entitled to access the segment.

2. Check if the segment type is appropriate to its intended use.

3. Check if the pointer violates the segment limit.

Although the 80386 processor automatically performs checks 2 and 3 during instruction execution, software must assist in performing the first check. The unprivileged instruction ARPL is provided for this purpose. Software can also explicitly perform steps 2 and 3 to check for potential violations (rather than waiting for an exception). The unprivileged instructions LAR, LSL, VERR, and VERW are provided for this purpose.

LAR (Load Access Rights) is used to verify that a pointer refers to a segment of the proper privilege level and type. LAR has one operand——a selector for a descriptor whose access rights are to be examined. The descriptor must be visible at the privilege level which is the maximum of the CPL and the selector's RPL. If the descriptor is visible, LAR obtains a masked form of the second doubleword of the descriptor, masks this value with 00FxFF00H, stores the result into the specified 32-bit destination register, and sets the zero flag. (The x indicates that the corresponding four bits of the stored value are undefined.) Once loaded, the access-rights bits can be tested. All valid descriptor types can be tested by the LAR instruction. If the RPL or CPL is greater than DPL, or if the selector is outside the table limit, no access-rights value is returned, and the zero flag is cleared. Conforming code segments may be accessed from any privilege level.

LSL (Load Segment Limit) allows software to test the limit of a descriptor. If the descriptor denoted by the given selector (in memory or a register) is visible at the CPL, LSL loads the specified 32-bit register with a 32-bit, byte granular, unscrambled limit that is calculated from fragmented limit fields and the G-bit of that descriptor. This can only be done for segments (data, code, task state, and local descriptor tables); gate descriptors are inaccessible. (Table 6-4 lists in detail which types are valid and which are not.) Interpreting the limit is a function of the segment type. For example, downward expandable data segments treat the limit differently than code segments do. For both LAR and LSL, the zero flag (ZF) is set if the loading was performed; otherwise, the ZF is cleared.

**Table 6-4. Valid Descriptor Types for LSL**

| Type Code | Descriptor Type | Valid? |
|---|---|---|
| 0 | (invalid) | NO |
| 1 | Available 286 TSS | YES |
| 2 | LDT | YES |
| 3 | Busy 286 TSS | YES |
| 4 | 286 Call Gate | NO |
| 5 | Task Gate | NO |
| 6 | 286 Trap Gate | NO |
| 7 | 286 Interrupt Gate | NO |
| 8 | (invalid) | NO |
| 9 | Available 386 TSS | YES |
| A | (invalid) | NO |
| B | Busy 386 TSS | YES |
| C | 386 Call Gate | NO |
| D | (invalid) | NO |
| E | 386 Trap Gate | NO |
| F | 386 Interrupt Gate | NO |

### 6.3.6.1 Descriptor Validation

The 80386 has two instructions, VERR and VERW, which determine whether a selector points to a segment that can be read or written at the current privilege level. Neither instruction causes a protection fault if the result is negative.

VERR (Verify for Reading) verifies a segment for reading and loads ZF with 1 if that segment is readable from the current privilege level. VERR checks that:

- The selector points to a descriptor within the bounds of the GDT or LDT.
- It denotes a code or data segment descriptor.
- The segment is readable and of appropriate privilege level.

The privilege check for data segments and nonconforming code segments is that the DPL must be numerically greater than or equal to both the CPL and the selector's RPL. Conforming segments are not checked for privilege level.

VERW (Verify for Writing) provides the same capability as VERR for verifying writability. Like the VERR instruction, VERW loads ZF if the result of the writability check is positive. The instruction checks that the descriptor is within bounds, is a segment descriptor, is writable, and that its DPL is numerically greater or equal to both the CPL and the selector's RPL. Code segments are never writable, conforming or not.

#### 6.3.6.2 Pointer Integrity and RPL

The Requestor's Privilege Level (RPL) feature can prevent inappropriate use of pointers that could corrupt the operation of more privileged code or data from a less privileged level.

A common example is a file system procedure, FREAD (file_id, n_bytes, buffer_ptr). This hypothetical procedure reads data from a file into a buffer, overwriting whatever is there. Normally, FREAD would be available at the user level, supplying only pointers to the file system procedures and data located and operating at a privileged level. Normally, such a procedure prevents user-level procedures from directly changing the file tables. However, in the absence of a standard protocol for checking pointer validity, a user-level procedure could supply a pointer into the file tables in place of its buffer pointer, causing the FREAD procedure to corrupt them unwittingly.

Use of RPL can avoid such problems. The RPL field allows a privilege attribute to be assigned to a selector. This privilege attribute would normally indicate the privilege level of the code which generated the selector. The 80386 processor automatically checks the RPL of any selector loaded into a segment register to determine whether the RPL allows access.

To take advantage of the processor's checking of RPL, the called procedure need only ensure that all selectors passed to it have an RPL at least as high (numerically) as the original caller's CPL. This action guarantees that selectors are not more trusted than their supplier. If one of the selectors is used to access a segment that the caller would not be able to access directly, i.e., the RPL is numerically greater than the DPL, then a protection fault will result when that selector is loaded into a segment register.

ARPL (Adjust Requestor's Privilege Level) adjusts the RPL field of a selector to become the larger of its original value and the value of the RPL field in a specified register. The latter is normally loaded from the image of the caller's CS register which is on the stack. If the adjustment changes the selector's RPL, ZF (the zero flag) is set; otherwise, ZF is cleared.

## 6.4 Page-Level Protection

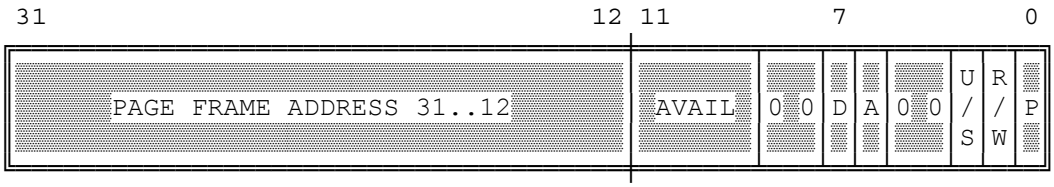
Two kinds of protection are related to pages:

1. Restriction of addressable domain.
2. Type checking.

### 6.4.1 Page-Table Entries Hold Protection Parameters

Figure 6-10 highlights the fields of PDEs and PTEs that control access to pages.

**Figure 6-10. Protection Fields of Page Table Entries**




R/W - READ/WRITE
U/S - USER/SUPERVISOR

### 6.4.1.1 Restricting Addressable Domain

The concept of privilege for pages is implemented by assigning each page to one of two levels:

1. Supervisor level (U/S=0) —— for the operating system and other systems software and related data.

2. User level (U/S=1) —— for applications procedures and data.

The current level (U or S) is related to CPL. If CPL is 0, 1, or 2, the processor is executing at supervisor level. If CPL is 3, the processor is executing at user level.

When the processor is executing at supervisor level, all pages are addressable, but, when the processor is executing at user level, only pages that belong to the user level are addressable.

### 6.4.1.2 Type Checking

At the level of page addressing, two types are defined:

1. Read-only access (R/W=0)
2. Read/write access (R/W=1)

When the processor is executing at supervisor level, all pages are both readable and writable. When the processor is executing at user level, only pages that belong to user level and are marked for read/write access are writable; pages that belong to supervisor level are neither readable nor writable from user level.

## 6.4.2 Combining Protection of Both Levels of Page Tables

For any one page, the protection attributes of its page directory entry may differ from those of its page table entry. The 80386 computes the effective protection attributes for a page by examining the protection attributes in both the directory and the page table. Table 6-5 shows the effective protection provided by the possible combinations of protection attributes.

### 6.4.3 Overrides to Page Protection

Certain accesses are checked as if they are privilege-level 0 references, even if CPL = 3:

- LDT, GDT, TSS, IDT references.
- Access to inner stack during ring-crossing CALL/INT.

## 6.5 Combining Page and Segment Protection

When paging is enabled, the 80386 first evaluates segment protection, then evaluates page protection. If the processor detects a protection violation at either the segment or the page level, the requested operation cannot proceed; a protection exception occurs instead.

For example, it is possible to define a large data segment which has some subunits that are read-only and other subunits that are read-write. In this case, the page directory (or page table) entries for the read-only subunits would have the U/S and R/W bits set to x0, indicating no write rights for all the pages described by that directory entry (or for individual pages). This technique might be used, for example, in a UNIX-like system to define a large data segment, part of which is read only (for shared data or ROMmed constants). This enables UNIX-like systems to define a "flat" data space as one large segment, use "flat" pointers to address within this "flat" space, yet be able to protect shared data, shared files mapped into the virtual space, and supervisor areas.

**Table 6-5. Combining Directory and Page Protection**

| Page Directory Entry | | Page Table Entry | | Combined Protection | |
|---|---|---|---|---|---|
| U/S | R/W | U/S | R/W | U/S | R/W |
| S-0 | R-0 | S-0 | R-0 | S | x |
| S-0 | R-0 | S-0 | W-1 | S | x |
| S-0 | R-0 | U-1 | R-0 | S | x |
| S-0 | R-0 | U-1 | W-1 | S | x |
| S-0 | W-1 | S-0 | R-0 | S | x |
| S-0 | W-1 | S-0 | W-1 | S | x |
| S-0 | W-1 | U-1 | R-0 | S | x |
| S-0 | W-1 | U-1 | W-1 | S | x |
| U-1 | R-0 | S-0 | R-0 | S | x |
| U-1 | R-0 | S-0 | W-1 | S | x |
| U-1 | R-0 | U-1 | R-0 | U | R |
| U-1 | R-0 | U-1 | W-1 | U | R |
| U-1 | W-1 | S-0 | R-0 | S | x |
| U-1 | W-1 | S-0 | W-1 | S | x |
| U-1 | W-1 | U-1 | R-0 | U | R |
| U-1 | W-1 | U-1 | W-1 | U | W |

NOTE
S —— Supervisor
R —— Read only
U —— User
W —— Read and Write
x indicates that when the combined U/S attribute is S, the R/W attribute is not checked.

# Chapter 7 Multitasking

To provide efficient, protected multitasking, the 80386 employs several special data structures. It does not, however, use special instructions to control multitasking; instead, it interprets ordinary control-transfer instructions differently when they refer to the special data structures. The registers and data structures that support multitasking are:

- Task state segment
- Task state segment descriptor
- Task register
- Task gate descriptor

With these structures the 80386 can rapidly switch execution from one task to another, saving the context of the original task so that the task can be restarted later. In addition to the simple task switch, the 80386 offers two other task-management features:

1. Interrupts and exceptions can cause task switches (if needed in the system design). The processor not only switches automatically to the task that handles the interrupt or exception, but it automatically switches back to the interrupted task when the interrupt or exception has been serviced. Interrupt tasks may interrupt lower-priority interrupt tasks to any depth.

2. With each switch to another task, the 80386 can also switch to another LDT and to another page directory. Thus each task can have a different logical-to-linear mapping and a different linear-to-physical mapping. This is yet another protection feature, because tasks can be isolated and prevented from interfering with one another.

## 7.1 Task State Segment

All the information the processor needs in order to manage a task is stored in a special type of segment, a task state segment (TSS). Figure 7-1 shows the format of a TSS for executing 80386 tasks. (Another format is used for executing 80286 tasks; refer to Chapter 13.)

The fields of a TSS belong to two classes:

1. A dynamic set that the processor updates with each switch from the task. This set includes the fields that store:

   - The general registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI).

   - The segment registers (ES, CS, SS, DS, FS, GS).

   - The flags register (EFLAGS).

   - The instruction pointer (EIP).

- The selector of the TSS of the previously executing task (updated only when a return is expected).

2. A static set that the processor reads but does not change. This set includes the fields that store:

   - The selector of the task's LDT.

   - The register (PDBR) that contains the base address of the task's page directory (read only when paging is enabled).

   - Pointers to the stacks for privilege levels 0-2.

   - The T-bit (debug trap bit) which causes the processor to raise a debug exception when a task switch occurs. (Refer to Chapter 12 for more information on debugging.)

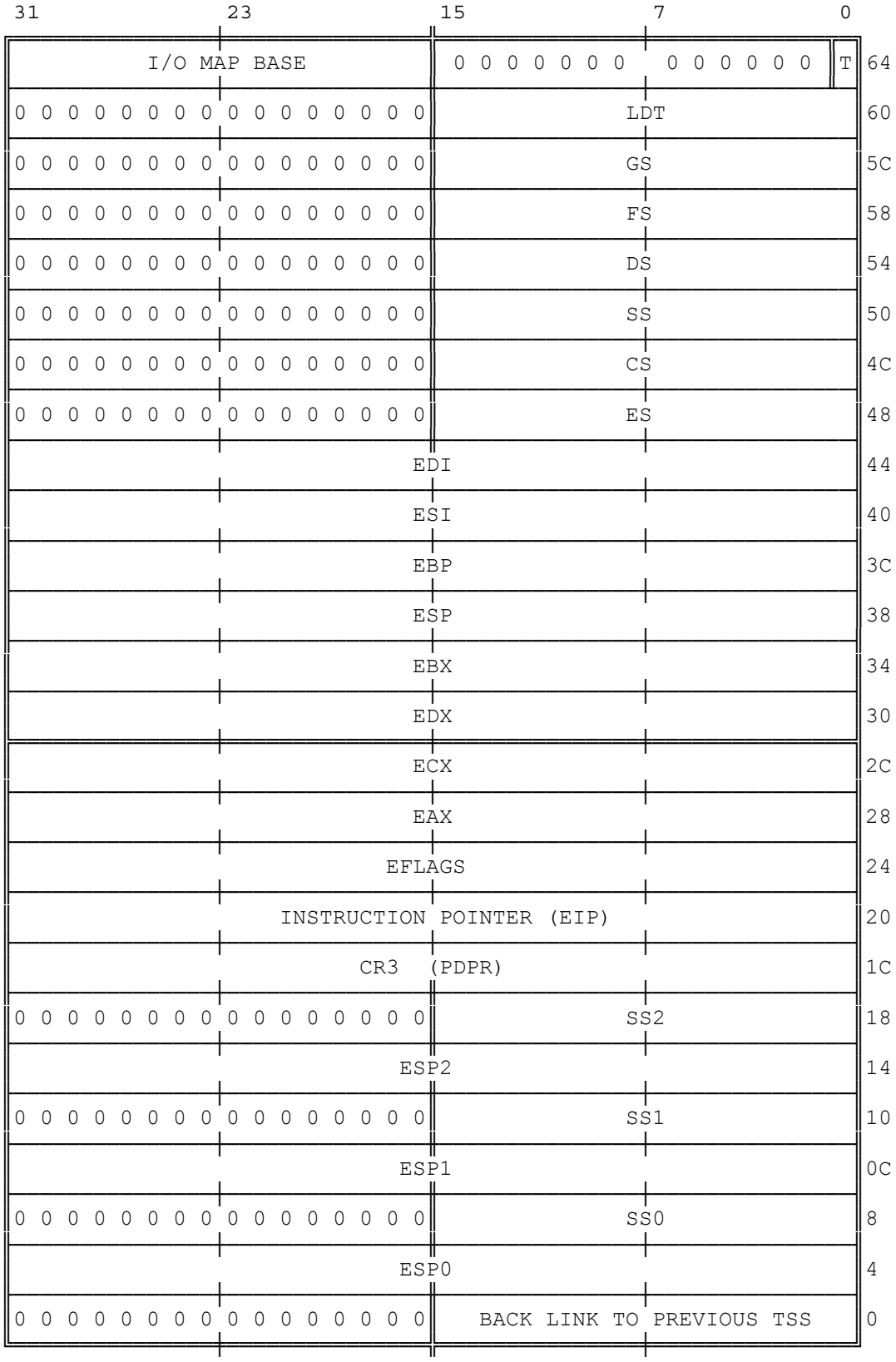
   - The I/O map base (refer to Chapter 8 for more information on the use of the I/O map).

Task state segments may reside anywhere in the linear space. The only case that requires caution is when the TSS spans a page boundary and the higher-addressed page is not present. In this case, the processor raises an exception if it encounters the not-present page while reading the TSS during a task switch. Such an exception can be avoided by either of two strategies:

1. By allocating the TSS so that it does not cross a page boundary.

2. By ensuring that both pages are either both present or both not-present at the time of a task switch. If both pages are not-present, then the page-fault handler must make both pages present before restarting the instruction that caused the task switch.

**Figure 7-1. 80386 32-Bit Task State Segment**



NOTE
0 MEANS INTEL RESERVED. DO NOT DEFINE.

## 7.2 TSS Descriptor

The task state segment, like all other segments, is defined by a descriptor. Figure 7-2 shows the format of a TSS descriptor.

The B-bit in the type field indicates whether the task is busy. A type code of 9 indicates a non-busy task; a type code of 11 indicates a busy task. Tasks are not reentrant. The B-bit allows the processor to detect an attempt to switch to a task that is already busy.

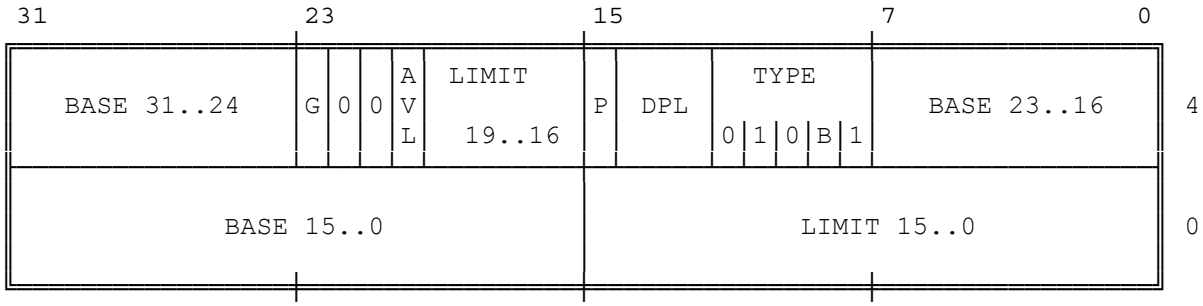
The BASE, LIMIT, and DPL fields and the G-bit and P-bit have functions similar to their counterparts in data-segment descriptors. The LIMIT field, however, must have a value equal to or greater than 103. An attempt to switch to a task whose TSS descriptor has a limit less that 103 causes an exception. A larger limit is permissible, and a larger limit is required if an I/O permission map is present. A larger limit may also be convenient for systems software if additional data is stored in the same segment as the TSS.

A procedure that has access to a TSS descriptor can cause a task switch. In most systems the DPL fields of TSS descriptors should be set to zero, so that only trusted software has the right to perform task switching.

Having access to a TSS-descriptor does not give a procedure the right to read or modify a TSS. Reading and modification can be accomplished only with another descriptor that redefines the TSS as a data segment. An attempt to load a TSS descriptor into any of the segment registers (CS, SS, DS, ES, FS, GS) causes an exception.

TSS descriptors may reside only in the GDT. An attempt to identify a TSS with a selector that has TI=1 (indicating the current LDT) results in an exception.

**Figure 7-2. TSS Descriptor for 32-bit TSS**



## 7.3 Task Register

The task register (TR) identifies the currently executing task by pointing to the TSS. Figure 7-3 shows the path by which the processor accesses the current TSS.

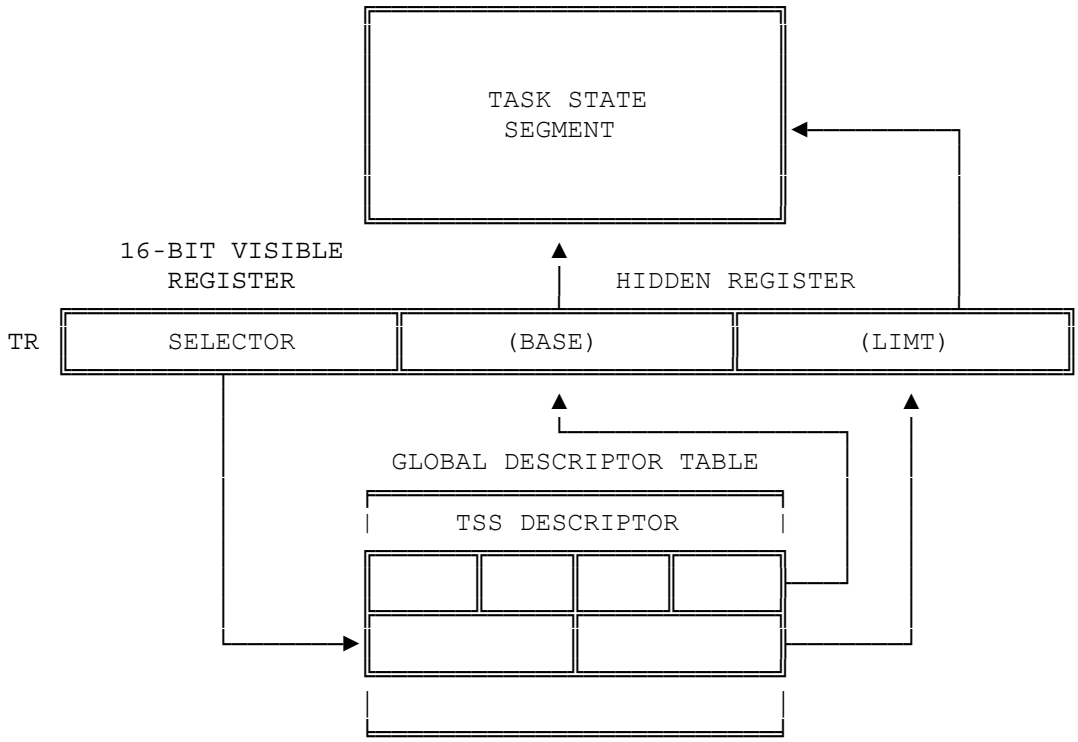
The task register has both a "visible" portion (i.e., can be read and changed by instructions) and an "invisible" portion (maintained by the processor to correspond to the visible portion; cannot be read by any instruction). The selector in the visible portion selects a TSS descriptor in the GDT. The processor uses the invisible portion to cache the base and limit values from the TSS descriptor. Holding the base and limit in a register makes execution of the task more efficient, because the processor does not need to repeatedly fetch these values from memory when it references the TSS of the current task.

The instructions LTR and STR are used to modify and read the visible portion of the task register. Both instructions take one operand, a 16-bit selector located in memory or in a general register.

LTR (Load task register) loads the visible portion of the task register with the selector operand, which must select a TSS descriptor in the GDT. LTR also loads the invisible portion with information from the TSS descriptor selected by the operand. LTR is a privileged instruction; it may be executed only when CPL is zero. LTR is generally used during system initialization to give an initial value to the task register; thereafter, the contents of TR are changed by task switch operations.

STR (Store task register) stores the visible portion of the task register in a general register or memory word. STR is not privileged.

**Figure 7-3. Task Register**




## 7.4 Task Gate Descriptor

A task gate descriptor provides an indirect, protected reference to a TSS. Figure 7-4 illustrates the format of a task gate.

The SELECTOR field of a task gate must refer to a TSS descriptor. The value of the RPL in this selector is not used by the processor.

The DPL field of a task gate controls the right to use the descriptor to cause a task switch. A procedure may not select a task gate descriptor unless the maximum of the selector's RPL and the CPL of the procedure is numerically less than or equal to the DPL of the descriptor. This constraint prevents untrusted procedures from causing a task switch. (Note that when a task gate is used, the DPL of the target TSS descriptor is not used for privilege checking.)

A procedure that has access to a task gate has the power to cause a task switch, just as a procedure that has access to a TSS descriptor. The 80386 has task gates in addition to TSS descriptors to satisfy three needs:

1. The need for a task to have a single busy bit. Because the busy-bit is stored in the TSS descriptor, each task should have only one such descriptor. There may, however, be several task gates that select the single TSS descriptor.

2. The need to provide selective access to tasks. Task gates fulfill this need, because they can reside in LDTs and can have a DPL that is different from the TSS descriptor's DPL. A procedure that does not have sufficient privilege to use the TSS descriptor in the GDT (which usually has a DPL of 0) can still switch to another task if it has access to a task gate for that task in its LDT. With task gates, systems software can limit the right to cause task switches to specific tasks.

3. The need for an interrupt or exception to cause a task switch. Task gates may also reside in the IDT, making it possible for interrupts and exceptions to cause task switching. When interrupt or exception vectors to an IDT entry that contains a task gate, the 80386 switches to the indicated task. Thus, all tasks in the system can benefit from the protection afforded by isolation from interrupt tasks.

Figure 7-5 illustrates how both a task gate in an LDT and a task gate in the IDT can identify the same task.

**Figure 7-4. Task Gate Descriptor**

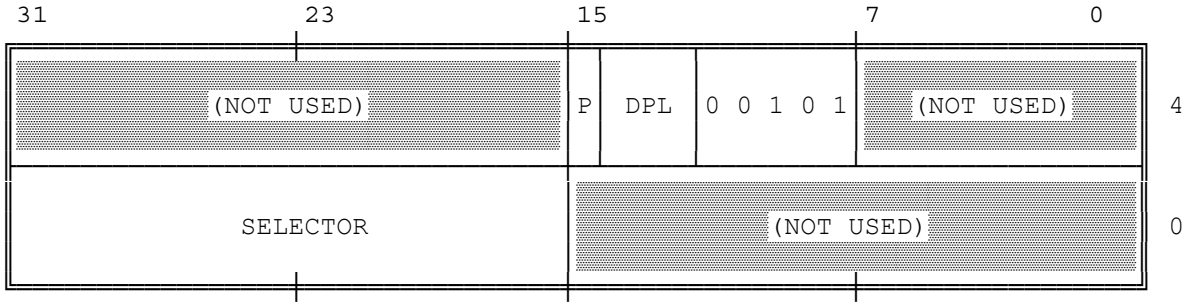
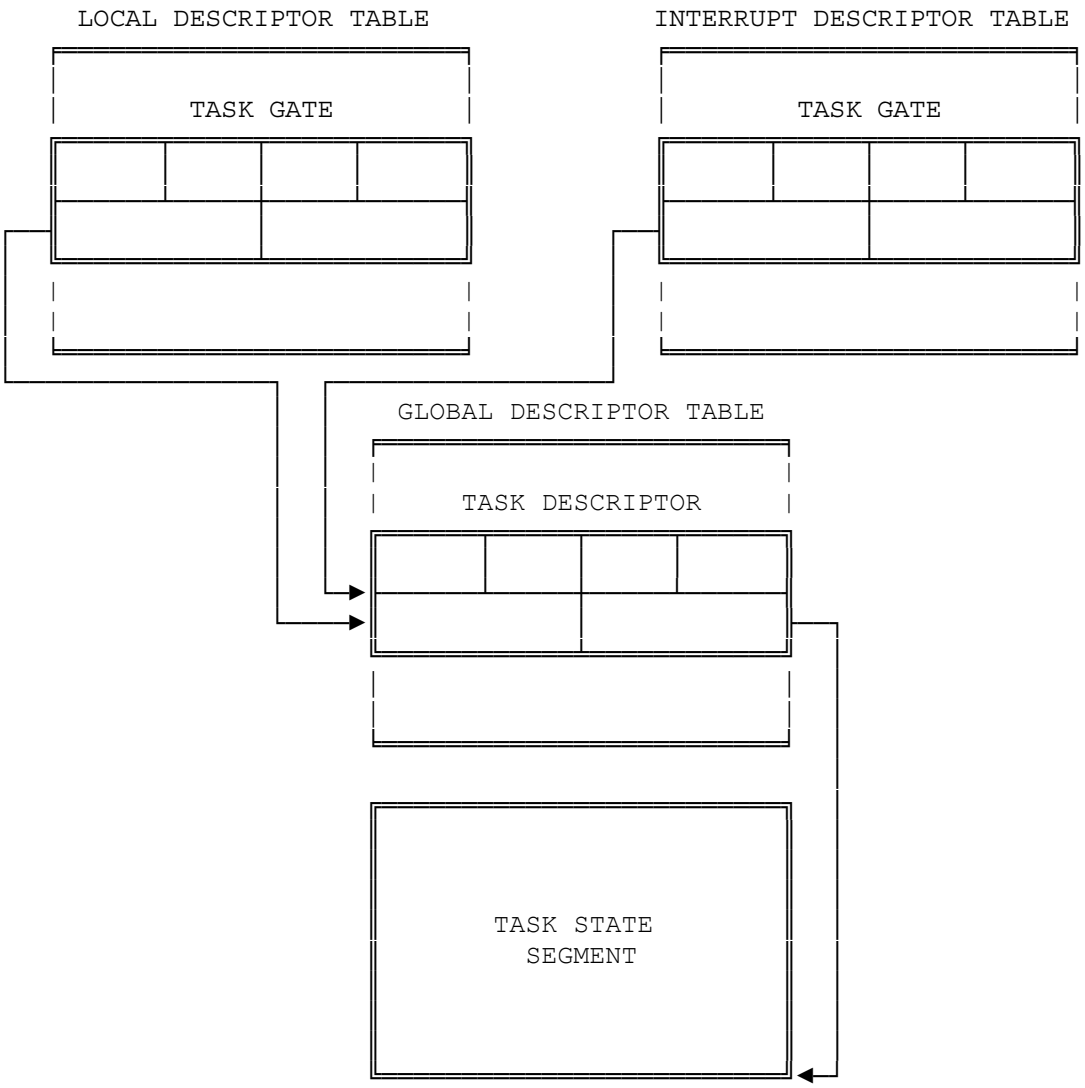


**Figure 7-5. Task Gate Indirectly Identifies Task**




## 7.5 Task Switching

The 80386 switches execution to another task in any of four cases:

1. The current task executes a JMP or CALL that refers to a TSS descriptor.

2. The current task executes a JMP or CALL that refers to a task gate.

3. An interrupt or exception vectors to a task gate in the IDT.

4. The current task executes an IRET when the NT flag is set.

JMP, CALL, IRET, interrupts, and exceptions are all ordinary mechanisms of the 80386 that can be used in circumstances that do not require a task

switch. Either the type of descriptor referenced or the NT (nested task) bit in the flag word distinguishes between the standard mechanism and the variant that causes a task switch.

To cause a task switch, a JMP or CALL instruction can refer either to a TSS descriptor or to a task gate. The effect is the same in either case: the 80386 switches to the indicated task.

An exception or interrupt causes a task switch when it vectors to a task gate in the IDT. If it vectors to an interrupt or trap gate in the IDT, a task switch does not occur. Refer to Chapter 9 for more information on the interrupt mechanism.

Whether invoked as a task or as a procedure of the interrupted task, an interrupt handler always returns control to the interrupted procedure in the interrupted task. If the NT flag is set, however, the handler is an interrupt task, and the IRET switches back to the interrupted task.

A task switching operation involves these steps:

1. Checking that the current task is allowed to switch to the designated task. Data-access privilege rules apply in the case of JMP or CALL instructions. The DPL of the TSS descriptor or task gate must be less than or equal to the maximum of CPL and the RPL of the gate selector. Exceptions, interrupts, and IRETs are permitted to switch tasks regardless of the DPL of the target task gate or TSS descriptor.

2. Checking that the TSS descriptor of the new task is marked present and has a valid limit. Any errors up to this point occur in the context of the outgoing task. Errors are restartable and can be handled in a way that is transparent to applications procedures.

3. Saving the state of the current task. The processor finds the base address of the current TSS cached in the task register. It copies the registers into the current TSS (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, ES, CS, SS, DS, FS, GS, and the flag register). The EIP field of the TSS points to the instruction after the one that caused the task switch.

4. Loading the task register with the selector of the incoming task's TSS descriptor, marking the incoming task's TSS descriptor as busy, and setting the TS (task switched) bit of the MSW. The selector is either the operand of a control transfer instruction or is taken from a task gate.

5. Loading the incoming task's state from its TSS and resuming execution. The registers loaded are the LDT register; the flag register; the general registers EIP, EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; the segment registers ES, CS, SS, DS, FS, and GS; and PDBR. Any errors detected in this step occur in the context of the incoming task. To an exception handler, it appears that the first instruction of the new task has not yet executed.

Note that the state of the outgoing task is always saved when a task switch occurs. If execution of that task is resumed, it starts after the instruction that caused the task switch. The registers are restored to the values they held when the task stopped executing.

Every task switch sets the TS (task switched) bit in the MSW (machine status word). The TS flag is useful to systems software when a coprocessor (such as a numerics coprocessor) is present. The TS bit signals that the context of the coprocessor may not correspond to the current 80386 task. Chapter 11 discusses the TS bit and coprocessors in more detail.

Exception handlers that field task-switch exceptions in the incoming task (exceptions due to tests 4 thru 16 of Table 7-1) should be cautious about taking any action that might load the selector that caused the exception. Such an action will probably cause another exception, unless the exception handler first examines the selector and fixes any potential problem.

The privilege level at which execution resumes in the incoming task is neither restricted nor affected by the privilege level at which the outgoing task was executing. Because the tasks are isolated by their separate address spaces and TSSs and because privilege rules can be used to prevent improper access to a TSS, no privilege rules are needed to constrain the relation between the CPLs of the tasks. The new task begins executing at the privilege level indicated by the RPL of the CS selector value that is loaded from the TSS.

**Table 7-1. Checks Made during a Task Switch**

| Test | Test Description | Exception | Error Code Selects |
|---|---|---|---|
| NP = Segment-not-present exception, GP = General protection fault, TS = Invalid TSS, SF = Stack fault | | | |
| 1 | Incoming TSS descriptor is present | NP | Incoming TSS |
| 2 | Incoming TSS descriptor is marked not-busy | GP | Incoming TSS |
| 3 | Limit of incoming TSS is greater than or equal to 103 | TS | Incoming TSS |
| | —— All register and selector values are loaded —— | | |
| 4 | LDT selector of incoming task is valid | TS | Incoming TSS |
| 5 | LDT of incoming task is present | TS | Incoming TSS |
| 6 | CS selector is valid | | |
| Validity tests of a selector check that the selector is in the proper table (eg., the LDT selector refers to the GDT), lies within the bounds of the table, and refers to the proper type of descriptor (e.g., the LDT selector refers to an LDT descriptor). | | TS | Code segment |
| 7 | Code segment is present | NP | Code segment |
| 8 | Code segment DPL matches CS RPL | TS | Code segment |
| 9 | Stack segment is valid | | |
| Validity tests of a selector check that the selector is in the proper table (eg., the LDT selector refers to the GDT), lies within the bounds of the table, and refers to the proper type of descriptor (e.g., the LDT selector refers to an LDT descriptor). | | GP | Stack segment |
| 10 | Stack segment is present | SF | Stack segment |
| 11 | Stack segment DPL = CPL | SF | Stack segment |
| 12 | Stack-selector RPL = CPL | GP | Stack segment |
| 13 | DS, ES, FS, GS selectors are valid | GP | Segment |
| Validity tests of a selector check that the selector is in the proper table (eg., the LDT selector refers to the GDT), lies within the bounds of the table, and refers to the proper type of descriptor (e.g., the LDT selector refers to an LDT descriptor). | | | |
| 14 | DS, ES, FS, GS segments are readable | GP | Segment |
| 15 | DS, ES, FS, GS segments are present | NP | Segment |
| 16 | DS, ES, FS, GS segment DPL ≥ CPL (unless these are conforming segments) | GP | Segment |

## 7.6 Task Linking

The back-link field of the TSS and the NT (nested task) bit of the flag word together allow the 80386 to automatically return to a task that CALLed another task or was interrupted by another task. When a CALL instruction, an interrupt instruction, an external interrupt, or an exception causes a switch to a new task, the 80386 automatically fills the back-link of the new TSS with the selector of the outgoing task's TSS and, at the same time, sets the NT bit in the new task's flag register. The NT flag indicates whether the back-link field is valid. The new task releases control by executing an IRET instruction. When interpreting an IRET, the 80386 examines the NT flag. If NT is set, the 80386 switches back to the task selected by the back-link field. Table 7-2 summarizes the uses of these fields.

**Table 7-2. Effect of Task Switch on BUSY, NT, and Back-Link**

| Affected Field | Effect of JMP Instruction | Effect of CALL Instruction | Effect of IRET Instruction |
|---|---|---|---|
| Busy bit of incoming task | Set, must be 0 before | Set, must be 0 before | Unchanged, must be set |
| Busy bit of outgoing task | Cleared | Unchanged (already set) | Cleared |
| NT bit of incoming task | Cleared | Set | Unchanged |
| NT bit of outgoing task | Unchanged | Unchanged | Cleared |
| Back-link of incoming task | Unchanged | Set to outgoing TSS selector | Unchanged |
| Back-link of outgoing task | Unchanged | Unchanged | Unchanged |

### 7.6.1 Busy Bit Prevents Loops

The B-bit (busy bit) of the TSS descriptor ensures the integrity of the back-link. A chain of back-links may grow to any length as interrupt tasks interrupt other interrupt tasks or as called tasks call other tasks. The busy bit ensures that the CPU can detect any attempt to create a loop. A loop would indicate an attempt to reenter a task that is already busy; however, the TSS is not a reentrable resource.

The processor uses the busy bit as follows:

1. When switching to a task, the processor automatically sets the busy bit of the new task.

2. When switching from a task, the processor automatically clears the busy bit of the old task if that task is not to be placed on the back-link chain (i.e., the instruction causing the task switch is JMP or IRET). If the task is placed on the back-link chain, its busy bit remains set.

3. When switching to a task, the processor signals an exception if the busy bit of the new task is already set.

By these actions, the processor prevents a task from switching to itself or to any task that is on a back-link chain, thereby preventing invalid reentry into a task.

The busy bit is effective even in multiprocessor configurations, because the processor automatically asserts a bus lock when it sets or clears the busy bit. This action ensures that two processors do not invoke the same task at the same time. (Refer to Chapter 11 for more on multiprocessing.)

### 7.6.2 Modifying Task Linkages

Any modification of the linkage order of tasks should be accomplished only by software that can be trusted to correctly update the back-link and the busy-bit. Such changes may be needed to resume an interrupted task before the task that interrupted it. Trusted software that removes a task from the back-link chain must follow one of the following policies:

1. First change the back-link field in the TSS of the interrupting task, then clear the busy-bit in the TSS descriptor of the task removed from the list.

2. Ensure that no interrupts occur between updating the back-link chain and the busy bit.

## 7.7 Task Address Space

The LDT selector and PDBR fields of the TSS give software systems designers flexibility in utilization of segment and page mapping features of the 80386. By appropriate choice of the segment and page mappings for each task, tasks may share address spaces, may have address spaces that are largely distinct from one another, or may have any degree of sharing between these two extremes.

The ability for tasks to have distinct address spaces is an important aspect of 80386 protection. A module in one task cannot interfere with a module in another task if the modules do not have access to the same address spaces. The flexible memory management features of the 80386 allow systems designers to assign areas of shared address space to those modules of different tasks that are designed to cooperate with each other.

### 7.7.1 Task Linear-to-Physical Space Mapping

The choices for arranging the linear-to-physical mappings of tasks fall into two general classes:

1. One linear-to-physical mapping shared among all tasks.

   When paging is not enabled, this is the only possibility. Without page tables, all linear addresses map to the same physical addresses.

   When paging is enabled, this style of linear-to-physical mapping results from using one page directory for all tasks. The linear space utilized may exceed the physical space available if the operating system also implements page-level virtual memory.

2. Several partially overlapping linear-to-physical mappings.

   This style is implemented by using a different page directory for each task. Because the PDBR (page directory base register) is loaded from the TSS with each task switch, each task may have a different page directory.

In theory, the linear address spaces of different tasks may map to completely distinct physical addresses. If the entries of different page directories point to different page tables and the page tables point to different pages of physical memory, then the tasks do not share any physical addresses.

In practice, some portion of the linear address spaces of all tasks must map to the same physical addresses. The task state segments must lie in a common space so that the mapping of TSS addresses does not change while the processor is reading and updating the TSSs during a task switch. The linear space mapped by the GDT should also be mapped to a common physical space; otherwise, the purpose of the GDT is defeated. Figure 7-6 shows how the linear spaces of two tasks can overlap in the physical space by sharing page tables.

### 7.7.2 Task Logical Address Space

By itself, a common linear-to-physical space mapping does not enable sharing of data among tasks. To share data, tasks must also have a common logical-to-linear space mapping; i.e., they must also have access to descriptors that point into a shared linear address space. There are three ways to create common logical-to-physical address-space mappings:

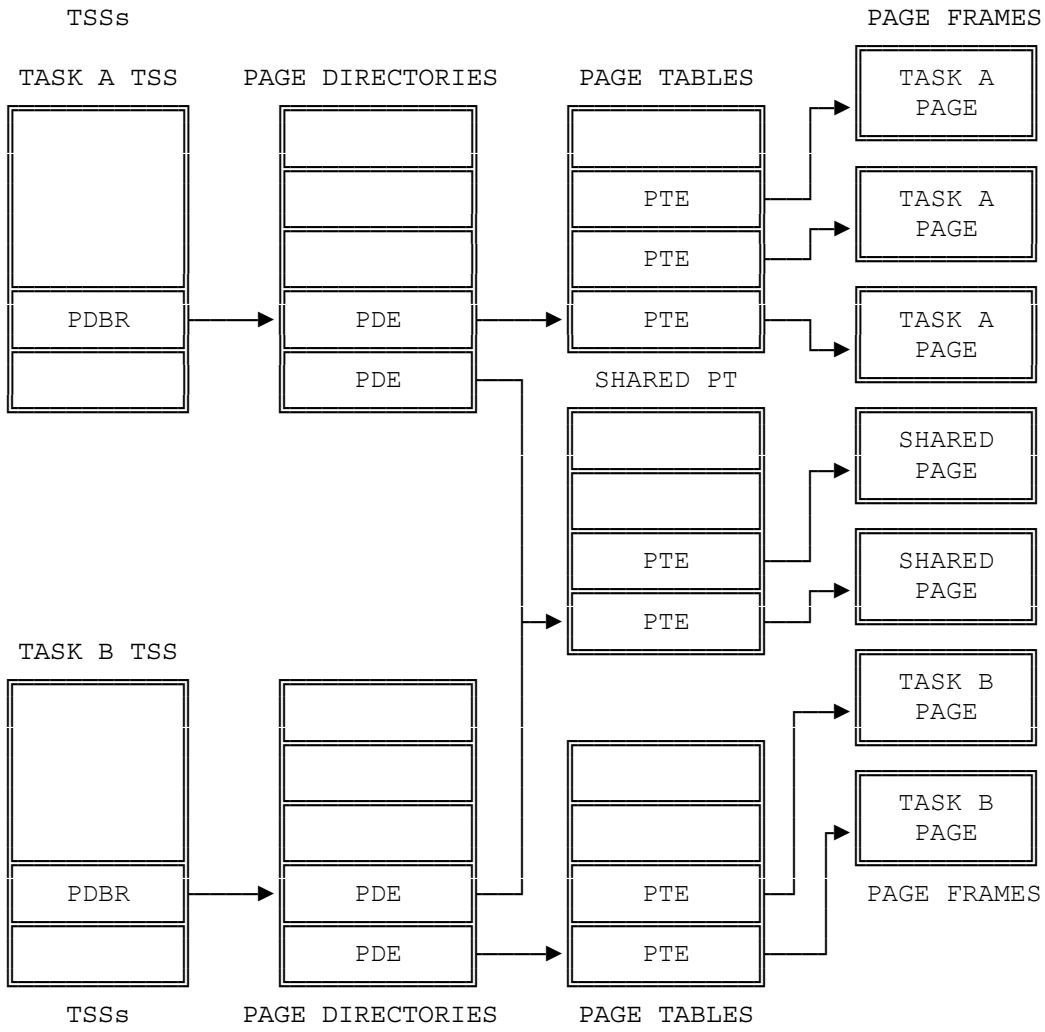
1. Via the GDT. All tasks have access to the descriptors in the GDT. If those descriptors point into a linear-address space that is mapped to a common physical-address space for all tasks, then the tasks can share data and instructions.

2. By sharing LDTs. Two or more tasks can use the same LDT if the LDT selectors in their TSSs select the same LDT segment. Those LDT-resident descriptors that point into a linear space that is mapped

to a common physical space permit the tasks to share physical memory. This method of sharing is more selective than sharing by the GDT; the sharing can be limited to specific tasks. Other tasks in the system may have different LDTs that do not give them access to the shared areas.

3. By descriptor aliases in LDTs. It is possible for certain descriptors of different LDTs to point to the same linear address space. If that linear address space is mapped to the same physical space by the page mapping of the tasks involved, these descriptors permit the tasks to share the common space. Such descriptors are commonly called "aliases". This method of sharing is even more selective than the prior two; other descriptors in the LDTs may point to distinct linear addresses or to linear addresses that are not shared.

**Figure 7-6. Partially-Overlapping Linear Spaces**



# Chapter 8 Input/Output

This chapter presents the I/O features of the 80386 from the following perspectives:

- Methods of addressing I/O ports
- Instructions that cause I/O operations
- Protection as it applies to the use of I/O instructions and I/O port addresses.

## 8.1 I/O Addressing

The 80386 allows input/output to be performed in either of two ways:

- By means of a separate I/O address space (using specific I/O instructions)
- By means of memory-mapped I/O (using general-purpose operand manipulationinstructions).

### 8.1.1 I/O Address Space

The 80386 provides a separate I/O address space, distinct from physical memory, that can be used to address the input/output ports that are used for external 16 devices. The I/O address space consists of $2^{16}$ (64K) individually addressable 8-bit ports; any two consecutive 8-bit ports can be treated as a 16-bit port; and four consecutive 8-bit ports can be treated as a 32-bit port. Thus, the I/O address space can accommodate up to 64K 8-bit ports, up to 32K 16-bit ports, or up to 16K 32-bit ports.

The program can specify the address of the port in two ways. Using an immediate byte constant, the program can specify:

- 256 8-bit ports numbered 0 through 255.
- 128 16-bit ports numbered 0, 2, 4, . . . , 252, 254.
- 64 32-bit ports numbered 0, 4, 8, . . . , 248, 252.

Using a value in DX, the program can specify:

- 8-bit ports numbered 0 through 65535
- 16-bit ports numbered 0, 2, 4, . . . , 65532, 65534
- 32-bit ports numbered 0, 4, 8, . . . , 65528, 65532

The 80386 can transfer 32, 16, or 8 bits at a time to a device located in the I/O space. Like doublewords in memory, 32-bit ports should be aligned at addresses evenly divisible by four so that the 32 bits can be transferred in

a single bus access. Like words in memory, 16-bit ports should be aligned at even-numbered addresses so that the 16 bits can be transferred in a single bus access. An 8-bit port may be located at either an even or odd address.

The instructions IN and OUT move data between a register and a port in the I/O address space. The instructions INS and OUTS move strings of data between the memory address space and ports in the I/O address space.

### 8.1.2 Memory-Mapped I/O

I/O devices also may be placed in the 80386 memory address space. As long as the devices respond like memory components, they are indistinguishable to the processor.

Memory-mapped I/O provides additional programming flexibility. Any instruction that references memory may be used to access an I/O port located in the memory space. For example, the MOV instruction can transfer data between any register and a port; and the AND, OR, and TEST instructions may be used to manipulate bits in the internal registers of a device (see Figure 8-1). Memory-mapped I/O performed via the full instruction set maintains the full complement of addressing modes for selecting the desired I/O device (e.g., direct address, indirect address, base register, index register, scaling).

Memory-mapped I/O, like any other memory reference, is subject to access protection and control when executing in protected mode. Refer to Chapter 6 for a discussion of memory protection.

## 8.2 I/O Instructions

The I/O instructions of the 80386 provide access to the processor's I/O ports for the transfer of data to and from peripheral devices. These instructions have as one operand the address of a port in the I/O address space. There are two classes of I/O instruction:

1. Those that transfer a single item (byte, word, or doubleword) located in a register.

2. Those that transfer strings of items (strings of bytes, words, or doublewords) located in memory. These are known as "string I/O instructions" or "block I/O instructions".

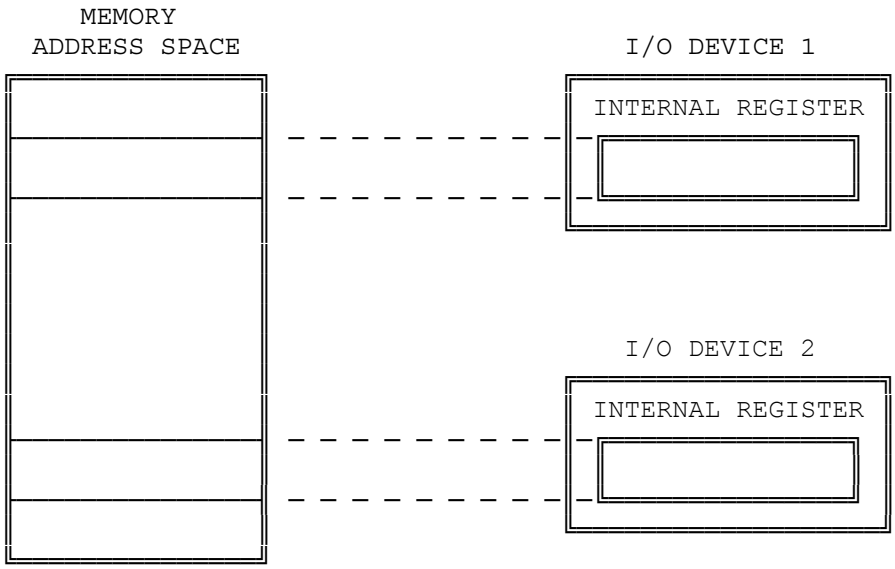
### 8.2.1 Register I/O Instructions

The I/O instructions IN and OUT are provided to move data between I/O ports and the EAX (32-bit I/O), the AX (16-bit I/O), or AL (8-bit I/O) general registers. IN and OUT instructions address I/O ports either directly, with the address of one of up to 256 port addresses coded in the instruction, or indirectly via the DX register to one of up to 64K port addresses.

IN (Input from Port) transfers a byte, word, or doubleword from an input port to AL, AX, or EAX. If a program specifies AL with the IN instruction, the processor transfers 8 bits from the selected port to AL. If a program specifies AX with the IN instruction, the processor transfers 16 bits from the port to AX. If a program specifies EAX with the IN instruction, the processor transfers 32 bits from the port to EAX.

OUT (Output to Port) transfers a byte, word, or doubleword to an output port from AL, AX, or EAX. The program can specify the number of the port using the same methods as the IN instruction.

**Figure 8-1. Memory-Mapped I/O**




## 8.2.2 Block I/O Instructions

The block (or string) I/O instructions INS and OUTS move blocks of data between I/O ports and memory space. Block I/O instructions use the DX register to specify the address of a port in the I/O address space. INS and OUTS use DX to specify:

- 8-bit ports numbered 0 through 65535
- 16-bit ports numbered 0, 2, 4, . . . , 65532, 65534
- 32-bit ports numbered 0, 4, 8, . . . , 65528, 65532

Block I/O instructions use either SI or DI to designate the source or destination memory address. For each transfer, SI or DI are automatically either incremented or decremented as specified by the direction bit in the flags register.

INS and OUTS, when used with repeat prefixes, cause block input or output operations. REP, the repeat prefix, modifies INS and OUTS to provide a means of transferring blocks of data between an I/O port and memory. These block

I/O instructions are string primitives (refer also to Chapter 3 for more on string primitives). They simplify programming and increase the speed of data transfer by eliminating the need to use a separate LOOP instruction or an intermediate register to hold the data.

The string I/O primitives can operate on byte strings, word strings, or doubleword strings. After each transfer, the memory address in ESI or EDI is updated by 1 for byte operands, by 2 for word operands, or by 4 for doubleword operands. The value in the direction flag (DF) determines whether the processor automatically increments ESI or EDI (DF=0) or whether it automatically decrements these registers (DF=1).

INS (Input String from Port) transfers a byte or a word string element from an input port to memory. The mnemonics INSB, INSW, and INSD are variants that explicitly specify the size of the operand. If a program specifies INSB, the processor transfers 8 bits from the selected port to the memory location indicated by ES:EDI. If a program specifies INSW, the processor transfers 16 bits from the port to the memory location indicated by ES:EDI. If a program specifies INSD, the processor transfers 32 bits from the port to the memory location indicated by ES:EDI. The destination segment register choice (ES) cannot be changed for the INS instruction. Combined with the REP prefix, INS moves a block of information from an input port to a series of consecutive memory locations.

OUTS (Output String to Port) transfers a byte, word, or doubleword string element to an output port from memory. The mnemonics OUTSB, OUTSW, and OUTSD are variants that explicitly specify the size of the operand. If a program specifies OUTSB, the processor transfers 8 bits from the memory location indicated by ES:EDI to the the selected port. If a program specifies OUTSW, the processor transfers 16 bits from the memory location indicated by ES:EDI to the the selected port. If a program specifies OUTSD, the processor transfers 32 bits from the memory location indicated by ES:EDI to the the selected port. Combined with the REP prefix, OUTS moves a block of information from a series of consecutive memory locations indicated by DS:ESI to an output port.

## 8.3 Protection and I/O

Two mechanisms provide protection for I/O functions:

1. The IOPL field in the EFLAGS register defines the right to use I/O-related instructions.

2. The I/O permission bit map of a 80386 TSS segment defines the right to use ports in the I/O address space.

These mechanisms operate only in protected mode, including virtual 8086 mode; they do not operate in real mode. In real mode, there is no protection of the I/O space; any procedure can execute I/O instructions, and any I/O port can be addressed by the I/O instructions.

### 8.3.1 I/O Privilege Level

Instructions that deal with I/O need to be restricted but also need to be executed by procedures executing at privilege levels other than zero. For this reason, the processor uses two bits of the flags register to store the I/O privilege level (IOPL). The IOPL defines the privilege level needed to execute I/O-related instructions.

The following instructions can be executed only if CPL ≤ IOPL:

IN —— Input
INS —— Input String
OUT —— Output
OUTS —— Output String
CLI —— Clear Interrupt-Enable Flag
STI —— Set Interrupt-Enable

These instructions are called "sensitive" instructions, because they are sensitive to IOPL.

To use sensitive instructions, a procedure must execute at a privilege level at least as privileged as that specified by the IOPL (CPL ≤ IOPL). Any attempt by a less privileged procedure to use a sensitive instruction results in a general protection exception.

Because each task has its own unique copy of the flags register, each task can have a different IOPL. A task whose primary function is to perform I/O (a device driver) can benefit from having an IOPL of three, thereby permitting all procedures of the task to performI/O. Other tasks typically have IOPL set to zero or one, reserving the right to perform I/O instructions for the most privileged procedures.

A task can change IOPL only with the POPF instruction; however, such changes are privileged. No procedure may alter IOPL (the I/O privilege level in the flag register) unless the procedure is executing at privilege level 0. An attempt by a less privileged procedure to alter IOPL does not result in an exception; IOPL simply remains unaltered.

The POPF instruction may be used in addition to CLI and STI to alter the interrupt-enable flag (IF); however, changes to IF by POPF are IOPL-sensitive. A procedure may alter IF with a POPF instruction only when executing at a level that is at least as privileged as IOPL. An attempt by a less privileged procedure to alter IF in this manner does not result in an exception; IF simply remains unaltered.

### 8.3.2 I/O Permission Bit Map

The I/O instructions that directly refer to addresses in the processor's I/O space are IN, INS, OUT, OUTS. The 80386 has the ability to selectively trap references to specific I/O addresses. The structure that enables selective trapping is the I/O Permission Bit Map in the TSS segment (see Figure 8-2). The I/O permission map is a bit vector. The size of the map and its location in the TSS segment are variable. The processor locates the

I/O permission map by means of the I/O map base field in the fixed portion of the TSS. The I/O map base field is 16 bits wide and contains the offset of the beginning of the I/O permission map. The upper limit of the I/O permission map is the same as the limit of the TSS segment.

In protected mode, when it encounters an I/O instruction (IN, INS, OUT, or OUTS), the processor first checks whether CPL ≤ IOPL. If this condition is true, the I/O operation may proceed. If not true, the processor checks the I/O permission map. (In virtual 8086 mode, the processor consults the map without regard for IOPL. Refer to Chapter 15.)

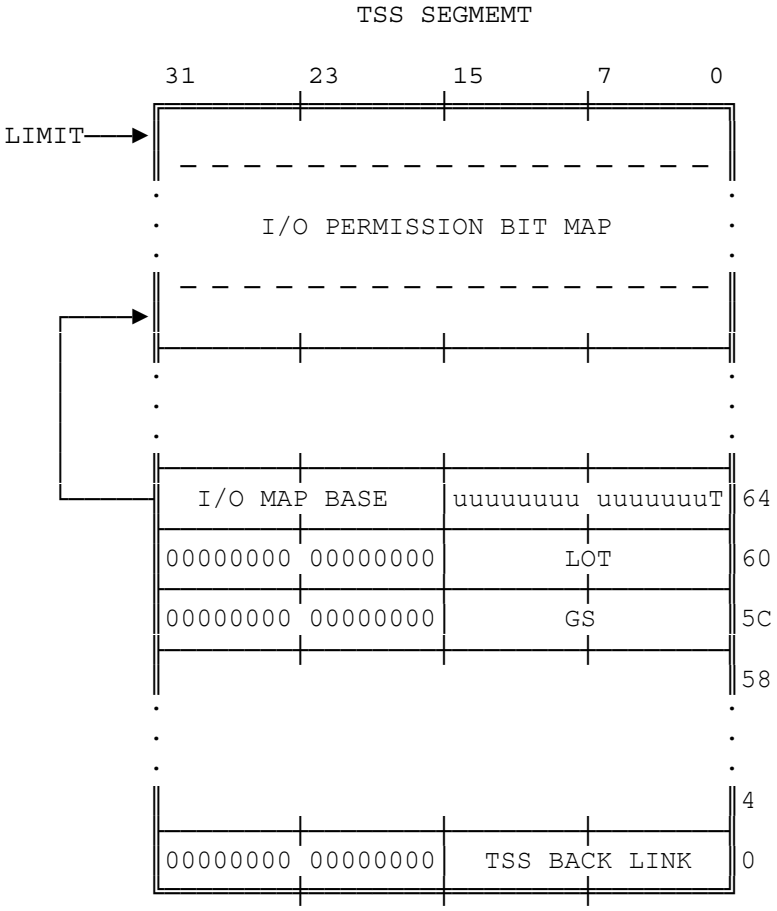
Each bit in the map corresponds to an I/O port byte address; for example, the bit for port 41 is found at I/O map base + 5, bit offset 1. The processor tests all the bits that correspond to the I/O addresses spanned by an I/O operation; for example, a doubleword operation tests four bits corresponding to four adjacent byte addresses. If any tested bit is set, the processor signals a general protection exception. If all the tested bits are zero, the I/O operation may proceed.

It is not necessary for the I/O permission map to represent all the I/O addresses. I/O addresses not spanned by the map are treated as if they had one bits in the map. For example, if TSS limit is equal to I/O map base + 31, the first 256 I/O ports are mapped; I/O operations on any port greater than 255 cause an exception.

If I/O map base is greater than or equal to TSS limit, the TSS segment has no I/O permission map, and all I/O instructions in the 80386 program cause exceptions when CPL > IOPL.

Because the I/O permission map is in the TSS segment, different tasks can have different maps. Thus, the operating system can allocate ports to a task by changing the I/O permission map in the task's TSS.

**Figure 8-2. I/O Address Bit Map**



# Chapter 9 Exceptions and Interrupts

Interrupts and exceptions are special kinds of control transfer; they work somewhat like unprogrammed CALLs. They alter the normal program flow to handle external events or to report errors or exceptional conditions. The difference between interrupts and exceptions is that interrupts are used to handle asynchronous events external to the processor, but exceptions handle conditions detected by the processor itself in the course of executing instructions.

There are two sources for external interrupts and two sources for exceptions:

1. Interrupts
    - Maskable interrupts, which are signalled via the INTR pin.
    - Nonmaskable interrupts, which are signalled via the NMI (Non-Maskable Interrupt) pin.
2. Exceptions
    - Processor detected. These are further classified as faults, traps, and aborts.
    - Programmed. The instructions INTO, INT 3, INT n, and BOUND can trigger exceptions. These instructions are often called "software interrupts", but the processor handles them as exceptions.

This chapter explains the features that the 80386 offers for controlling and responding to interrupts when it is executing in protected mode.

## 9.1 Identifying Interrupts

The processor associates an identifying number with each different type of interrupt or exception.

The NMI and the exceptions recognized by the processor are assigned predetermined identifiers in the range 0 through 31. Not all of these numbers are currently used by the 80386; unassigned identifiers in this range are reserved by Intel for possible future expansion.

The identifiers of the maskable interrupts are determined by external interrupt controllers (such as Intel's 8259A Programmable Interrupt Controller) and communicated to the processor during the processor's interrupt-acknowledge sequence. The numbers assigned by an 8259A PIC can be specified by software. Any numbers in the range 32 through 255 can be used. Table 9-1 shows the assignment of interrupt and exception identifiers.

Exceptions are classified as faults, traps, or aborts depending on the way they are reported and whether restart of the instruction that caused the exception is supported.

Faults Faults are exceptions that are reported "before" the instruction causingthe exception. Faults are either detected before the instruction begins to execute, or during execution of the instruction. If detected during the instruction, the fault is reported with the machine restored to a state that permits the instruction to be restarted.

Traps A trap is an exception that is reported at the instruction boundary immediately after the instruction in which the exception was detected.

Aborts An abort is an exception that permits neither precise location of the instruction causing the exception nor restart of the program that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

**Table 9-1. Interrupt and Exception ID Assignments**

| Identifier | Description |
|---|---|
| 0 | Divide error |
| 1 | Debug exceptions |
| 2 | Nonmaskable interrupt |
| 3 | Breakpoint (one-byte INT 3 instruction) |
| 4 | Overflow (INTO instruction) |
| 5 | Bounds check (BOUND instruction) |
| 6 | Invalid opcode |
| 7 | Coprocessor not available |
| 8 | Double fault |
| 9 | (reserved) |
| 10 | Invalid TSS |
| 11 | Segment not present |
| 12 | Stack exception |
| 13 | General protection |
| 14 | Page fault |
| 15 | (reserved) |
| 16 | Coprecessor error |
| 17-31 | (reserved) |
| 32-255 | Available for external interrupts via INTR pin |

## 9.2 Enabling and Disabling Interrupts

The processor services interrupts and exceptions only between the end of one instruction and the beginning of the next. When the repeat prefix is used to repeat a string instruction, interrupts and exceptions may occur between repetitions. Thus, operations on long strings do not delay interrupt response.

Certain conditions and flag settings cause the processor to inhibit certain interrupts and exceptions at instruction boundaries.

### 9.2.1 NMI Masks Further NMIs

While an NMI handler is executing, the processor ignores further interrupt signals at the NMI pin until the next IRET instruction is executed.

### 9.2.2 IF Masks INTR

The IF (interrupt-enable flag) controls the acceptance of external interrupts signalled via the INTR pin. When IF=0, INTR interrupts are inhibited; when IF=1, INTR interrupts are enabled. As with the other flag bits, the processor clears IF in response to a RESET signal. The instructions CLI and STI alter the setting of IF.

CLI (Clear Interrupt-Enable Flag) and STI (Set Interrupt-Enable Flag) explicitly alter IF (bit 9 in the flag register). These instructions may be executed only if CPL ≤ IOPL. A protection exception occurs if they are executed when CPL > IOPL.

The IF is also affected implicitly by the following operations:

- The instruction PUSHF stores all flags, including IF, in the stack where they can be examined.
- Task switches and the instructions POPF and IRET load the flags register; therefore, they can be used to modify IF.
- Interrupts through interrupt gates automatically reset IF, disabling interrupts. (Interrupt gates are explained later in this chapter.)

### 9.2.3 RF Masks Debug Faults

The RF bit in EFLAGS controls the recognition of debug faults. This permits debug faults to be raised for a given instruction at most once, no matter how many times the instruction is restarted. (Refer to Chapter 12 for more information on debugging.)

### 9.2.4 MOV or POP to SS Masks Some Interrupts and Exceptions

Software that needs to change stack segments often uses a pair of instructions; for example:

```
MOV SS, AX
MOV ESP, StackTop
```

If an interrupt or exception is processed after SS has been changed but before ESP has received the corresponding change, the two parts of the stack pointer SS:ESP are inconsistent for the duration of the interrupt handler or exception handler.

To prevent this situation, the 80386, after both a MOV to SS and a POP to SS instruction, inhibits NMI, INTR, debug exceptions, and single-step traps at the instruction boundary following the instruction that changes SS. Some exceptions may still occur; namely, page fault and general protection fault. Always use the 80386 LSS instruction, and the problem will not occur.

## 9.3 Priority Among Simultaneous Interrupts and Exceptions

If more than one interrupt or exception is pending at an instruction boundary, the processor services one of them at a time. The priority among classes of interrupt and exception sources is shown in Table 9-2. The processor first services a pending interrupt or exception from the class that has the highest priority, transferring control to the first instruction of the interrupt handler. Lower priority exceptions are discarded; lower priority interrupts are held pending. Discarded exceptions will be rediscovered when the interrupt handler returns control to the point of interruption.

## 9.4 Interrupt Descriptor Table

The interrupt descriptor table (IDT) associates each interrupt or exception identifier with a descriptor for the instructions that service the associated event. Like the GDT and LDTs, the IDT is an array of 8-byte descriptors. Unlike the GDT and LDTs, the first entry of the IDT may contain a descriptor. To form an index into the IDT, the processor multiplies the interrupt or exception identifier by eight. Because there are only 256 identifiers, the IDT need not contain more than 256 descriptors. It can contain fewer than 256 entries; entries are required only for interrupt identifiers that are actually used.

The IDT may reside anywhere in physical memory. As Figure 9-1 shows, the processor locates the IDT by means of the IDT register (IDTR). The instructions LIDT and SIDT operate on the IDTR. Both instructions have one explicit operand: the address in memory of a 6-byte area. Figure 9-2 shows the format of this area.

LIDT (Load IDT register) loads the IDT register with the linear base address and limit values contained in the memory operand. This instruction can be executed only when the CPL is zero. It is normally used by the initialization logic of an operating system when creating an IDT. An operating system may also use it to change from one IDT to another.

SIDT (Store IDT register) copies the base and limit value stored in IDTR to a memory location. This instruction can be executed at any privilege level.

**Table 9-2. Priority Among Simultaneous Interrupts and Exceptions**

| Priority | Class of Interrupt or Exception |
|---|---|
| HIGHEST | Faults except debug faults |
| | Trap instructions INTO, INT n, INT 3 |
| | Debug traps for this instruction |
| | Debug faults for next instruction |
| | NMI interrupt |
| LOWEST | INTR interrupt |

**Figure 9-1. IDT Register and Table**

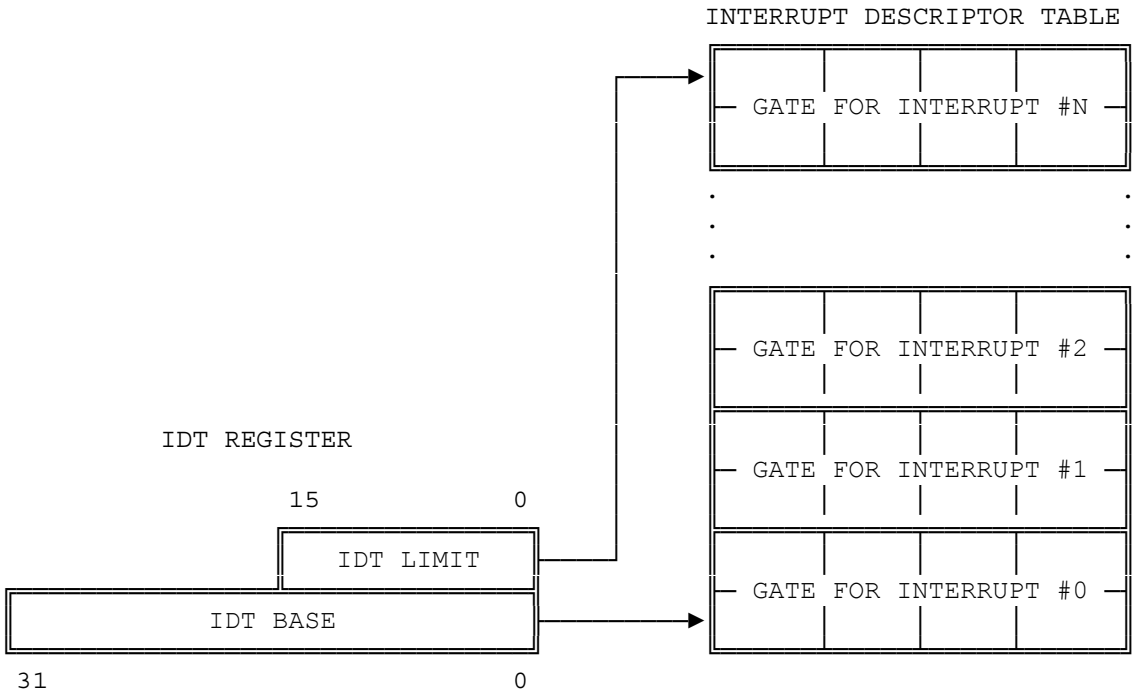
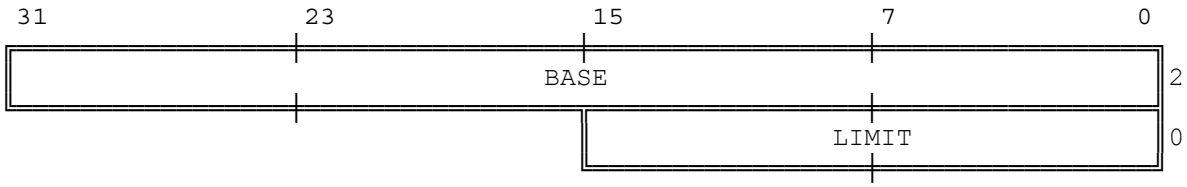



**Figure 9-2. Pseudo-Descriptor Format for LIDT and SIDT**



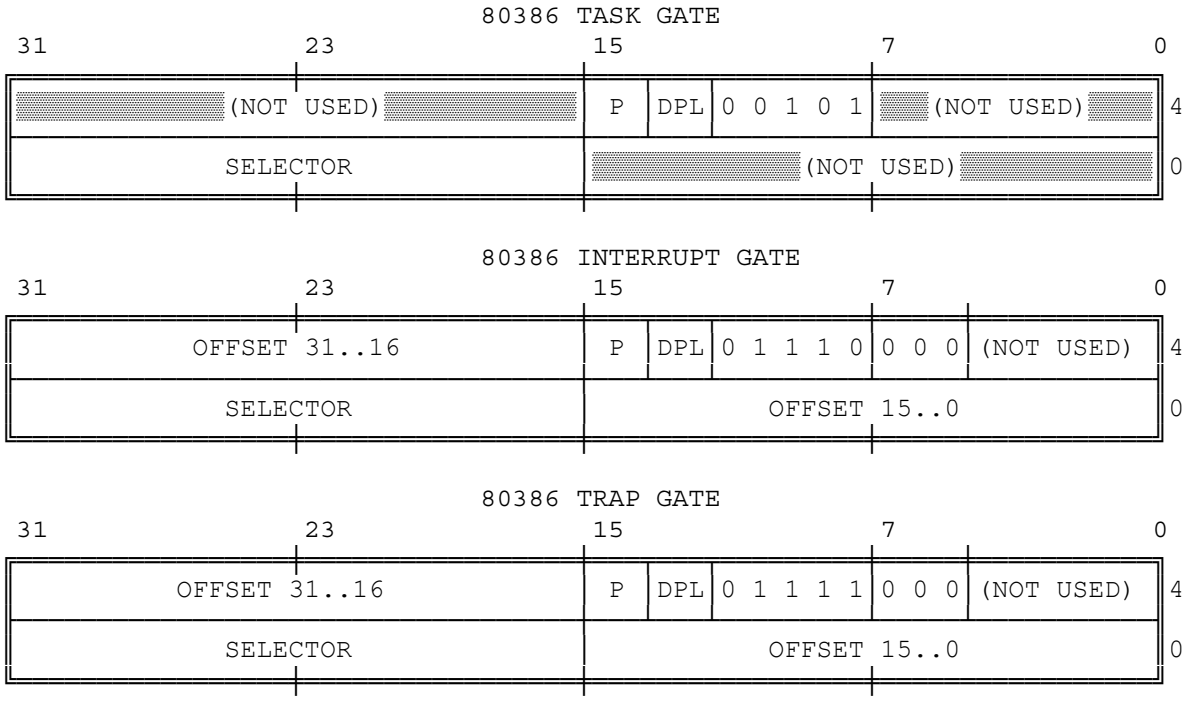
## 9.5 IDT Descriptors

The IDT may contain any of three kinds of descriptor:

- Task gates
- Interrupt gates
- Trap gates

Figure 9-3 illustrates the format of task gates and 80386 interrupt gates and trap gates. (The task gate in an IDT is the same as the task gate already discussed in Chapter 7.)

**Figure 9-3. 80306 IDT Gate Descriptors**




## 9.6 Interrupt Tasks and Interrupt Procedures

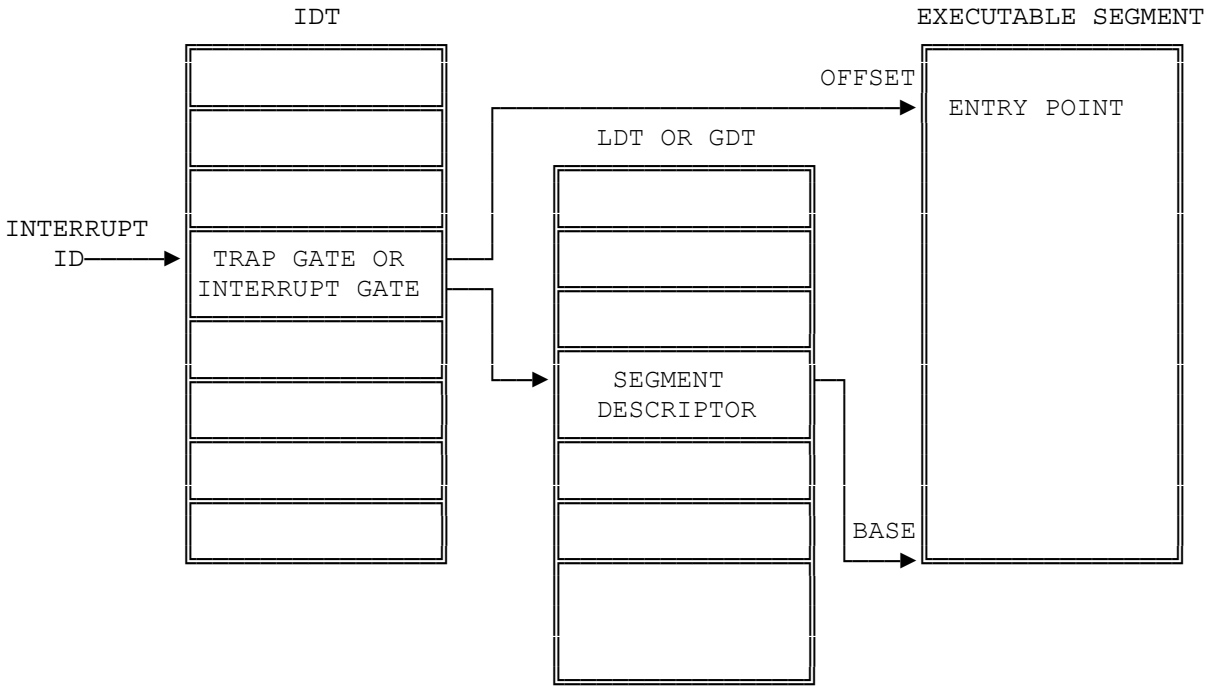
Just as a CALL instruction can call either a procedure or a task, so an interrupt or exception can "call" an interrupt handler that is either a procedure or a task. When responding to an interrupt or exception, the processor uses the interrupt or exception identifier to index a descriptor in the IDT. If the processor indexes to an interrupt gate or trap gate, it invokes the handler in a manner similar to a CALL to a call gate. If the processor finds a task gate, it causes a task switch in a manner similar to a CALL to a task gate.

### 9.6.1 Interrupt Procedures

An interrupt gate or trap gate points indirectly to a procedure which will execute in the context of the currently executing task as illustrated by Figure 9-4. The selector of the gate points to an executable-segment descriptor in either the GDT or the current LDT. The offset field of the gate points to the beginning of the interrupt or exception handling procedure.

The 80386 invokes an interrupt or exception handling procedure in much the same manner as it CALLs a procedure; the differences are explained in the following sections.

**Figure 9-4. Interrupt Vectoring for Procedures**




#### 9.6.1.1 Stack of Interrupt Procedure

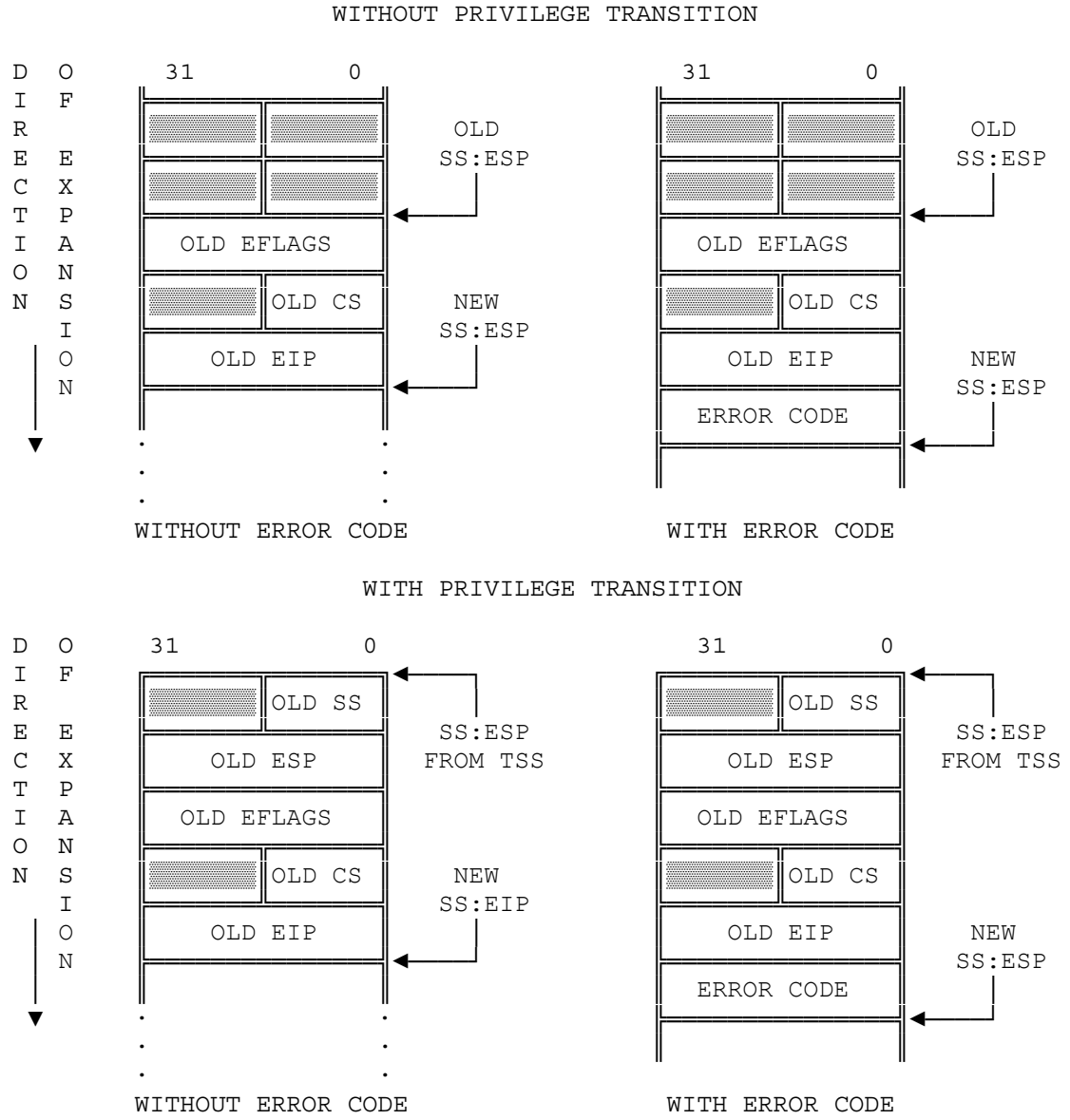
Just as with a control transfer due to a CALL instruction, a control transfer to an interrupt or exception handling procedure uses the stack to store the information needed for returning to the original procedure. As Figure 9-5 shows, an interrupt pushes the EFLAGS register onto the stack before the pointer to the interrupted instruction.

Certain types of exceptions also cause an error code to be pushed on the stack. An exception handler can use the error code to help diagnose the exception.

### 9.6.1.2 Returning from an Interrupt Procedure

An interrupt procedure also differs from a normal procedure in the method of leaving the procedure. The IRET instruction is used to exit from an interrupt procedure. IRET is similar to RET except that IRET increments EIP by an extra four bytes (because of the flags on the stack) and moves the saved flags into the EFLAGS register. The IOPL field of EFLAGS is changed only if the CPL is zero. The IF flag is changed only if CPL ≤ IOPL.

**Figure 9-5. Stack Layout after Exception of Interrupt**



#### 9.6.1.3 Flags Usage by Interrupt Procedure

Interrupts that vector through either interrupt gates or trap gates cause TF (the trap flag) to be reset after the current value of TF is saved on the stack as part of EFLAGS. By this action the processor prevents debugging activity that uses single-stepping from affecting interrupt response. A subsequent IRET instruction restores TF to the value in the EFLAGS image on the stack.

The difference between an interrupt gate and a trap gate is in the effect on IF (the interrupt-enable flag). An interrupt that vectors through an interrupt gate resets IF, thereby preventing other interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores IF to the value in the EFLAGS image on the stack. An interrupt through a trap gate does not change IF.

#### 9.6.1.4 Protection in Interrupt Procedures

The privilege rule that governs interrupt procedures is similar to that for procedure calls: the CPU does not permit an interrupt to transfer control to a procedure in a segment of lesser privilege (numerically greater privilege level) than the current privilege level. An attempt to violate this rule results in a general protection exception.

Because occurrence of interrupts is not generally predictable, this privilege rule effectively imposes restrictions on the privilege levels at which interrupt and exception handling procedures can execute. Either of the following strategies can be employed to ensure that the privilege rule is never violated.

- Place the handler in a conforming segment. This strategy suits the handlers for certain exceptions (divide error, for example). Such a handler must use only the data available to it from the stack. If it needed data from a data segment, the data segment would have to have privilege level three, thereby making it unprotected.

- Place the handler procedure in a privilege level zero segment.

### 9.6.2 Interrupt Tasks

A task gate in the IDT points indirectly to a task, as Figure 9-6 illustrates. The selector of the gate points to a TSS descriptor in the GDT.

When an interrupt or exception vectors to a task gate in the IDT, a task switch results. Handling an interrupt with a separate task offers two advantages:

- The entire context is saved automatically.

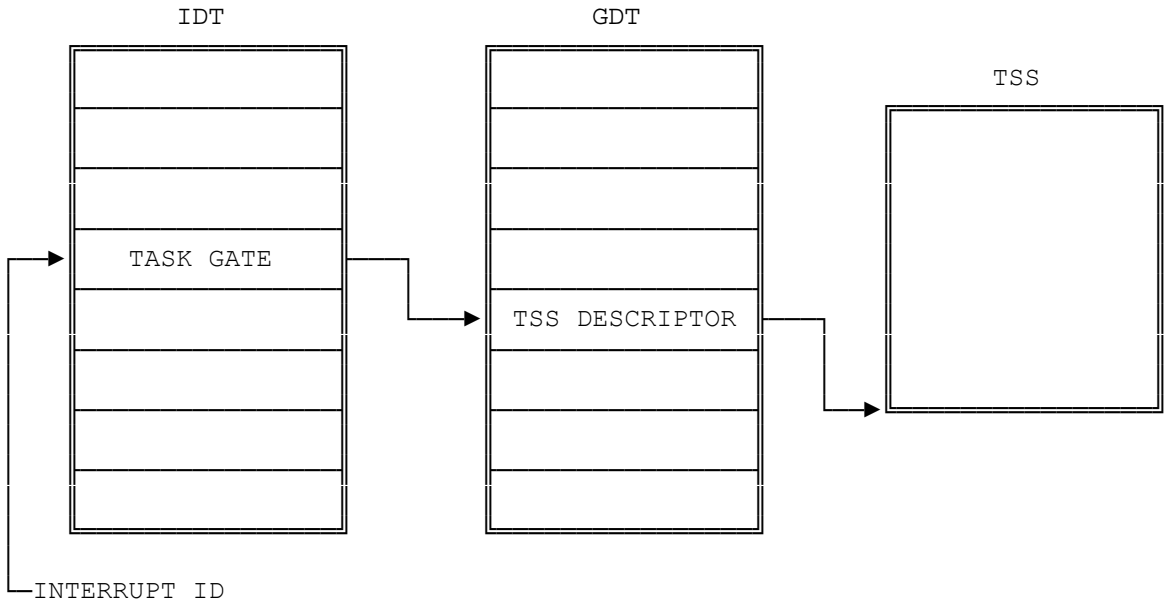
- The interrupt handler can be isolated from other tasks by giving it a separate address space, either via its LDT or via its page directory.

The actions that the processor takes to perform a task switch are discussed in Chapter 7. The interrupt task returns to the interrupted task by executing an IRET instruction.

If the task switch is caused by an exception that has an error code, the processor automatically pushes the error code onto the stack that corresponds to the privilege level of the first instruction to be executed in the interrupt task.

When interrupt tasks are used in an operating system for the 80386, there are actually two schedulers: the software scheduler (part of the operating system) and the hardware scheduler (part of the processor's interrupt mechanism). The design of the software scheduler should account for the fact that the hardware scheduler may dispatch an interrupt task whenever interrupts are enabled.

**Figure 9-6. Interrupt Vectoring for Tasks**




## 9.7 Error Code

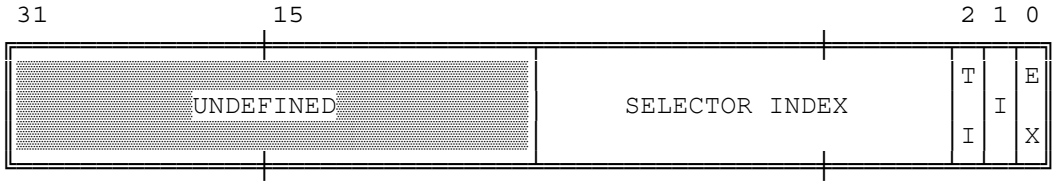
With exceptions that relate to a specific segment, the processor pushes an error code onto the stack of the exception handler (whether procedure or task). The error code has the format shown in Figure 9-7. The format of the error code resembles that of a selector; however, instead of an RPL field, the error code contains two one-bit items:

1. The processor sets the EXT bit if an event external to the program caused the exception.

2. The processor sets the I-bit (IDT-bit) if the index portion of the error code refers to a gate descriptor in the IDT.

If the I-bit is not set, the TI bit indicates whether the error code refers to the GDT (value 0) or to the LDT (value 1). The remaining 14 bits are the upper 14 bits of the segment selector involved. In some cases the error code on the stack is null, i.e., all bits in the low-order word are zero.

**Figure 9-7. Error Code Format**




## 9.8 Exception Conditions

The following sections describe each of the possible exception conditions in detail. Each description classifies the exception as a fault, trap, or abort. This classification provides information needed by systems programmers for restarting the procedure in which the exception occurred:

Faults The CS and EIP values saved when a fault is reported point to the instruction causing the fault.

Traps The CS and EIP values stored when the trap is reported point to the instruction dynamically after the instruction causing the trap. If a trap is detected during an instruction that alters program flow, the reported values of CS and EIP reflect the alteration of program flow. For example, if a trap is detected in a JMP instruction, the CS and EIP values pushed onto the stack point to the target of the JMP, not to the instruction after the JMP.

Aborts An abort is an exception that permits neither precise location of the instruction causing the exception nor restart of the program that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

### 9.8.1 Interrupt 0 —— Divide Error

The divide-error fault occurs during a DIV or an IDIV instruction when the divisor is zero.

### 9.8.2 Interrupt 1 —— Debug Exceptions

The processor triggers this interrupt for any of a number of conditions; whether the exception is a fault or a trap depends on the condition:

- Instruction address breakpoint fault.
- Data address breakpoint trap.
- General detect fault.
- Single-step trap.
- Task-switch breakpoint trap.

The processor does not push an error code for this exception. An exception handler can examine the debug registers to determine which condition caused the exception. Refer to Chapter 12 for more detailed information about debugging and the debug registers.

### 9.8.3 Interrupt 3 —— Breakpoint

The INT 3 instruction causes this trap. The INT 3 instruction is one byte long, which makes it easy to replace an opcode in an executable segment with the breakpoint opcode. The operating system or a debugging subsystem can use a data-segment alias for an executable segment to place an INT 3 anywhere it is convenient to arrest normal execution so that some sort of special processing can be performed. Debuggers typically use breakpoints as a way of displaying registers, variables, etc., at crucial points in a task.

The saved CS:EIP value points to the byte following the breakpoint. If a debugger replaces a planted breakpoint with a valid opcode, it must subtract one from the saved EIP value before returning. Refer also to Chapter 12 for more information on debugging.

### 9.8.4 Interrupt 4 —— Overflow

This trap occurs when the processor encounters an INTO instruction and the OF (overflow) flag is set. Since signed arithmetic and unsigned arithmetic both use the same arithmetic instructions, the processor cannot determine which is intended and therefore does not cause overflow exceptions automatically. Instead it merely sets OF when the results, if interpreted as signed numbers, would be out of range. When doing arithmetic on signed operands, careful programmers and compilers either test OF directly or use the INTO instruction.

### 9.8.5 Interrupt 5 —— Bounds Check

This fault occurs when the processor, while executing a BOUND instruction, finds that the operand exceeds the specified limits. A program can use the BOUND instruction to check a signed array index against signed limits defined in a block of memory.

### 9.8.6 Interrupt 6 —— Invalid Opcode

This fault occurs when an invalid opcode is detected by the execution unit. (The exception is not detected until an attempt is made to execute the invalid opcode; i.e., prefetching an invalid opcode does not cause this exception.) No error code is pushed on the stack. The exception can be handled within the same task.

This exception also occurs when the type of operand is invalid for the given opcode. Examples include an intersegment JMP referencing a register operand, or an LES instruction with a register source operand.

### 9.8.7 Interrupt 7 —— Coprocessor Not Available

This exception occurs in either of two conditions:

- The processor encounters an ESC (escape) instruction, and the EM (emulate) bit ofCR0 (control register zero) is set.
- The processor encounters either the WAIT instruction or an ESC instruction, and both the MP (monitor coprocessor) and TS (task switched) bits of CR0 are set.

Refer to Chapter 11 for information about the coprocessor interface.

### 9.8.8 Interrupt 8 —— Double Fault

Normally, when the processor detects an exception while trying to invoke the handler for a prior exception, the two exceptions can be handled serially. If, however, the processor cannot handle them serially, it signals the double-fault exception instead. To determine when two faults are to be signalled as a double fault, the 80386 divides the exceptions into three classes: benign exceptions, contributory exceptions, and page faults. Table 9-3 shows this classification.

Table 9-4 shows which combinations of exceptions cause a double fault and which do not.

The processor always pushes an error code onto the stack of the double-fault handler; however, the error code is always zero. The faulting instruction may not be restarted. If any other exception occurs while attempting to invoke the double-fault handler, the processor shuts down.

**Table 9-3. Double-Fault Detection Classes**

| Class | ID | Description |
|---|---|---|
| Benign Exceptions | 1 | Debug exceptions |
| | 2 | NMI |
| | 3 | Breakpoint |
| | 4 | Overflow |
| | 5 | Bounds check |
| | 6 | Invalid opcode |
| | 7 | Coprocessor not available |
| | 16 | Coprocessor error |
| Contributory Exceptions | 0 | Divide error |
| | 9 | Coprocessor Segment Overrun |
| | 10 | Invalid TSS |
| | 11 | Segment not present |
| | 12 | Stack exception |
| | 13 | General protection |
| Page Faults | 14 | Page fault |

**Table 9-4. Double-Fault Definition**

| | | SECOND EXCEPTION | | |
|---|---|---|---|---|
| | | Benign Exception | Contributory Exception | Page Fault |
| FIRST EXCEPTION | Benign Exception | OK | OK | OK |
| | Contributory Exception | OK | DOUBLE | OK |
| | Page Fault | OK | DOUBLE | DOUBLE |

### 9.8.9 Interrupt 9 —— Coprocessor Segment Overrun

This exception is raised in protected mode if the 80386 detects a page or segment violation while transferring the middle portion of a coprocessor operand to the NPX. This exception is avoidable. Refer to Chapter 11 for more information about the coprocessor interface.

### 9.8.10 Interrupt 10 —— Invalid TSS

Interrupt 10 occurs if during a task switch the new TSS is invalid. A TSS is considered invalid in the cases shown in Table 9-5. An error code is pushed onto the stack to help identify the cause of the fault. The EXT bit indicates whether the exception was caused by a condition outside the control of the program; e.g., an external interrupt via a task gate triggered a switch to an invalid TSS.

This fault can occur either in the context of the original task or in the context of the new task. Until the processor has completely verified the presence of the new TSS, the exception occurs in the context of the original task. Once the existence of the new TSS is verified, the task switch is considered complete; i.e., TR is updated and, if the switch is due to a CALL or interrupt, the backlink of the new TSS is set to the old TSS. Any errors discovered by the processor after this point are handled in the context of the new task.

To insure a proper TSS to process it, the handler for exception 10 must be a task invoked via a task gate.

**Table 9-5. Conditions That Invalidate the TSS**

| Error Code | Condition |
|---|---|
| TSS id + EXT | The limit in the TSS descriptor is less than 103 |
| LTD id + EXT | Invalid LDT selector or LDT not present |
| SS id + EXT | Stack segment selector is outside table limit |
| SS id + EXT | Stack segment is not a writable segment |
| SS id + EXT | Stack segment DPL does not match new CPL |
| SS id + EXT | Stack segment selector RPL < > CPL |
| CS id + EXT | Code segment selector is outside table limit |
| CS id + EXT | Code segment selector does not refer to code segment |
| CS id + EXT | DPL of non-conforming code segment < > new CPL |
| CS id + EXT | DPL of conforming code segment > new CPL |
| DS/ES/FS/GS id + EXT | DS, ES, FS, or GS segment selector is outside table limits |
| DS/ES/FS/GS id + EXT | DS, ES, FS, or GS is not readable segment |

### 9.8.11 Interrupt 11 —— Segment Not Present

Exception 11 occurs when the processor detects that the present bit of a descriptor is zero. The processor can trigger this fault in any of these cases:

- While attempting to load the CS, DS, ES, FS, or GS registers; loading the SS register, however, causes a stack fault.
- While attempting loading the LDT register with an LLDT instruction; loading the LDT register during a task switch operation, however, causes the "invalid TSS" exception.
- While attempting to use a gate descriptor that is marked not-present.

This fault is restartable. If the exception handler makes the segment present and returns, the interrupted program will resume execution.

If a not-present exception occurs during a task switch, not all the steps of the task switch are complete. During a task switch, the processor first loads all the segment registers, then checks their contents for validity. If a not-present exception is discovered, the remaining segment registers have

not been checked and therefore may not be usable for referencing memory. The not-present handler should not rely on being able to use the values found in CS, SS, DS, ES, FS, and GS without causing another exception. The exception handler should check all segment registers before trying to resume the new task; otherwise, general protection faults may result later under conditions that make diagnosis more difficult. There are three ways to handle this case:

1. Handle the not-present fault with a task. The task switch back to the interrupted task will cause the processor to check the registers as it loads them from the TSS.

2. PUSH and POP all segment registers. Each POP causes the processor to check the new contents of the segment register.

3. Scrutinize the contents of each segment-register image in the TSS, simulating the test that the processor makes when it loads a segment register.

This exception pushes an error code onto the stack. The EXT bit of the error code is set if an event external to the program caused an interrupt that subsequently referenced a not-present segment. The I-bit is set if the error code refers to an IDT entry, e.g., an INT instruction referencing a not-present gate.

An operating system typically uses the "segment not present" exception to implement virtual memory at the segment level. A not-present indication in a gate descriptor, however, usually does not indicate that a segment is not present (because gates do not necessarily correspond to segments). Not-present gates may be used by an operating system to trigger exceptions of special significance to the operating system.

### 9.8.12 Interrupt 12 —— Stack Exception

A stack fault occurs in either of two general conditions:

- As a result of a limit violation in any operation that refers to the SS register. This includes stack-oriented instructions such as POP, PUSH, ENTER, and LEAVE, as well as other memory references that implicitly use SS (for example, MOV AX, [BP+6]). ENTER causes this exception when the stack is too small for the indicated local-variable space.

- When attempting to load the SS register with a descriptor that is marked not-present but is otherwise valid. This can occur in a task switch, an interlevel CALL, an interlevel return, an LSS instruction, or a MOV or POP instruction to SS.

When the processor detects a stack exception, it pushes an error code onto the stack of the exception handler. If the exception is due to a not-present stack segment or to overflow of the new stack during an interlevel CALL, the error code contains a selector to the segment in question (the exception handler can test the present bit in the descriptor to determine which exception occurred); otherwise the error code is zero.

An instruction that causes this fault is restartable in all cases. The return pointer pushed onto the exception handler's stack points to the instruction that needs to be restarted. This instruction is usually the one that caused the exception; however, in the case of a stack exception due to loading of a not-present stack-segment descriptor during a task switch, the indicated instruction is the first instruction of the new task.

When a stack fault occurs during a task switch, the segment registers may not be usable for referencing memory. During a task switch, the selector values are loaded before the descriptors are checked. If a stack fault is discovered, the remaining segment registers have not been checked and therefore may not be usable for referencing memory. The stack fault handler should not rely on being able to use the values found in CS, SS, DS, ES, FS, and GS without causing another exception. The exception handler should check all segment registers before trying to resume the new task; otherwise, general protection faults may result later under conditions that make diagnosis more difficult.

### 9.8.13 Interrupt 13 —— General Protection Exception

All protection violations that do not cause another exception cause a general protection exception. This includes (but is not limited to):

1. Exceeding segment limit when using CS, DS, ES, FS, or GS
2. Exceeding segment limit when referencing a descriptor table
3. Transferring control to a segment that is not executable
4. Writing into a read-only data segment or into a code segment
5. Reading from an execute-only segment
6. Loading the SS register with a read-only descriptor (unless the selector comes from the TSS during a task switch, in which case a TSS exception occurs
7. Loading SS, DS, ES, FS, or GS with the descriptor of a system segment
8. Loading DS, ES, FS, or GS with the descriptor of an executable segment that is not also readable
9. Loading SS with the descriptor of an executable segment
10. Accessing memory via DS, ES, FS, or GS when the segment register contains a null selector
11. Switching to a busy task
12. Violating privilege rules
13. Loading CR0 with PG=1 and PE=0.
14. Interrupt or exception via trap or interrupt gate from V86 mode to privilege level other than zero.

15. Exceeding the instruction length limit of 15 bytes (this can occur only if redundant prefixes are placed before an instruction)

The general protection exception is a fault. In response to a general protection exception, the processor pushes an error code onto the exception handler's stack. If loading a descriptor causes the exception, the error code contains a selector to the descriptor; otherwise, the error code is null. The source of the selector in an error code may be any of the following:

1. An operand of the instruction.
2. A selector from a gate that is the operand of the instruction.
3. A selector from a TSS involved in a task switch.

### 9.8.14 Interrupt 14 —— Page Fault

This exception occurs when paging is enabled (PG=1) and the processor detects one of the following conditions while translating a linear address to a physical address:

- The page-directory or page-table entry needed for the address translation has zero in its present bit.

- The current procedure does not have sufficient privilege to access the indicated page.

The processor makes available to the page fault handler two items of information that aid in diagnosing the exception and recovering from it:

- An error code on the stack. The error code for a page fault has a format different from that for other exceptions (see Figure 9-8). The error code tells the exception handler three things:

  1. Whether the exception was due to a not present page or to an access rights violation.

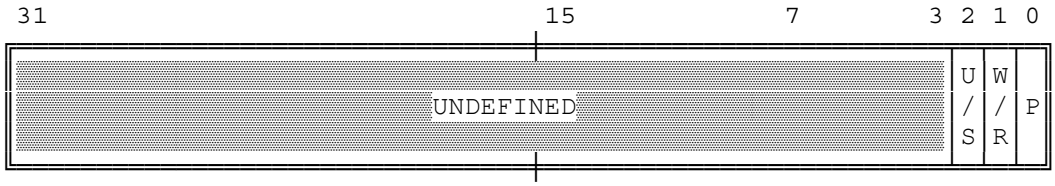
  2. Whether the processor was executing at user or supervisor level at the time of the exception.

  3. Whether the memory access that caused the exception was a read or write.

- CR2 (control register two). The processor stores in CR2 the linear address used in the access that caused the exception (see Figure 9-9). The exception handler can use this address to locate the corresponding page directory and page table entries. If another page fault can occur during execution of the page fault handler, the handler should push CR2 onto the stack.

**Figure 9-8. Page-Fault Error Code Format**

| Field | Value | Description |
|---|---|---|
| U/S | 0 | The access causing the fault originated when the processor was executing in supervisor mode. |
| | 1 | The access causing the fault originated when the processor was executing in user mode. |
| W/R | 0 | The access causing the fault was a read. |
| | 1 | The access causing the fault was a write. |
| P | 0 | The fault was caused by a not-present page. |
| | 1 | The fault was caused by a page-level protection violation. |



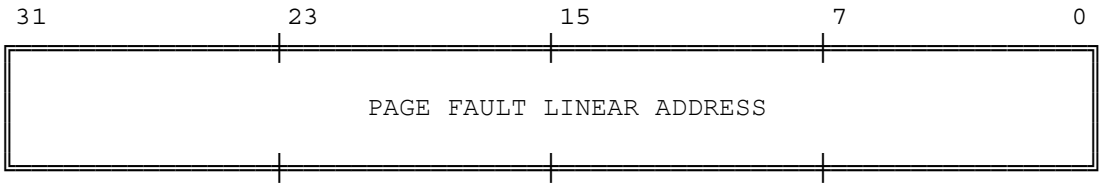

### 9.8.14.1 Page Fault During Task Switch

The processor may access any of four segments during a task switch:

1. Writes the state of the original task in the TSS of that task.
2. Reads the GDT to locate the TSS descriptor of the new task.
3. Reads the TSS of the new task to check the types of segment descriptors from the TSS.
4. May read the LDT of the new task in order to verify the segment registers stored in the new TSS.

A page fault can result from accessing any of these segments. In the latter two cases the exception occurs in the context of the new task. The instruction pointer refers to the next instruction of the new task, not to the instruction that caused the task switch. If the design of the operating system permits page faults to occur during task-switches, the page-fault handler should be invoked via a task gate.

**Figure 9-9. CR2 Format**




### 9.8.14.2 Page Fault with Inconsistent Stack Pointer

Special care should be taken to ensure that a page fault does not cause the processor to use an invalid stack pointer (SS:ESP). Software written for earlier processors in the 8086 family often uses a pair of instructions to change to a new stack; for example:

```
MOV SS, AX
MOV SP, StackTop
```

With the 80386, because the second instruction accesses memory, it is possible to get a page fault after SS has been changed but before SP has received the corresponding change. At this point, the two parts of the stack pointer SS:SP (or, for 32-bit programs, SS:ESP) are inconsistent.

The processor does not use the inconsistent stack pointer if the handling of the page fault causes a stack switch to a well defined stack (i.e., the handler is a task or a more privileged procedure). However, if the page fault handler is invoked by a trap or interrupt gate and the page fault occurs at the same privilege level as the page fault handler, the processor will attempt to use the stack indicated by the current (invalid) stack pointer.

In systems that implement paging and that handle page faults within the faulting task (with trap or interrupt gates), software that executes at the same privilege level as the page fault handler should initialize a new stack by using the new LSS instruction rather than an instruction pair shown above. When the page fault handler executes at privilege level zero (the normal case), the scope of the problem is limited to privilege-level zero code, typically the kernel of the operating system.

## 9.8.15 Interrupt 16 —— Coprocessor Error

The 80386 reports this exception when it detects a signal from the 80287 or 80387 on the 80386's ERROR# input pin. The 80386 tests this pin only at the beginning of certain ESC instructions and when it encounters a WAIT instruction while the EM bit of the MSW is zero (no emulation). Refer to Chapter 11 for more information on the coprocessor interface.

## 9.9 Exception Summary

Table 9-6 summarizes the exceptions recognized by the 386.

**Table 9-6. Exception Summary**

| Description | Interrupt Number | Return Address Points to Faulting Instruction | Exception Type | Function That Can Generate the Exception |
|---|---|---|---|---|
| Divide error | 0 | YES | FAULT | DIV, IDIV |
| Debug exceptions | 1 | Some debug exceptions are traps and some are faults. The exception handler can determine which has occurred by examining DR6. (Refer to Chapter 12.) | Some debug exceptions are traps and some are faults. The exception handler can determine which has occurred by examining DR6. (Refer to Chapter 12.) | Any instruction |
| Breakpoint | 3 | NO | TRAP | One-byte INT 3 |
| Overflow | 4 | NO | TRAP | INTO |
| Bounds check | 5 | YES | FAULT | BOUND |
| Invalid opcode | 6 | YES | FAULT | Any illegal instruction |
| Coprocessor not available | 7 | YES | FAULT | ESC, WAIT |
| Double fault | 8 | YES | ABORT | Any instruction that can generate an exception |
| Coprocessor Segment Overrun | 9 | NO | ABORT | Any operand of an ESC instruction that wraps around the end of a segment. |
| Invalid TSS | 10 | YES | FAULT | An invalid-TSS fault is not restartable if it occurs during the processing of an external interrupt. JMP, CALL, IRET, any interrupt |
| Segment not present | 11 | YES | FAULT | Any segment-register modifier |
| Stack exception | 12 | YES | FAULT | Any memory reference thru SS |
| General Protection | 13 | YES | FAULT/ABORT | All GP faults are restartable. If the fault occurs while attempting to vector to the handler for an external interrupt, the interrupted program is restartable, but the interrupt may be lost. Any memory reference or code fetch |

| | | | | |
|---|---|---|---|---|
| Page fault | 14 | YES | FAULT | Any memory reference or code fetch |
| Coprocessor error | 16 | YES | FAULT | Coprocessor errors are reported as a fault on the first ESC or WAIT instruction executed after the ESC instruction that caused the error. ESC, WAIT |
| Two-byte SW Interrupt | 0-255 | NO | TRAP | INT n |

## 9.10 Error Code Summary

Table 9-7 summarizes the error information that is available with each exception.

**Table 9-7. Error-Code Summary**

| Description | Interrupt Number | Error Code |
|---|---|---|
| Divide error | 0 | No |
| Debug exceptions | 1 | No |
| Breakpoint | 3 | No |
| Overflow | 4 | No |
| Bounds check | 5 | No |
| Invalid opcode | 6 | No |
| Coprocessor not available | 7 | No |
| System error | 8 | Yes (always 0) |
| Coprocessor Segment Overrun | 9 | No |
| Invalid TSS | 10 | Yes |
| Segment not present | 11 | Yes |
| Stack exception | 12 | Yes |
| General protection fault | 13 | Yes |
| Page fault | 14 | Yes |
| Coprocessor error | 16 | No |
| Two-byte SW interrupt | 0-255 | No |

# Chapter 10 Initialization

After a signal on the RESET pin, certain registers of the 80386 are set to predefined values. These values are adequate to enable execution of a bootstrap program, but additional initialization must be performed by software before all the features of the processor can be utilized.

## 10.1 Processor State After Reset

The contents of EAX depend upon the results of the power-up self test. The self-test may be requested externally by assertion of BUSY# at the end of RESET. The EAX register holds zero if the 80386 passed the test. A nonzero value in EAX after self-test indicates that the particular 80386 unit is faulty. If the self-test is not requested, the contents of EAX after RESET is undefined.

DX holds a component identifier and revision number after RESET as Figure 10-1 illustrates. DH contains 3, which indicates an 80386 component. DL contains a unique identifier of the revision level.

Control register zero (CR0) contains the values shown in Figure 10-2. The ET bit of CR0 is set if an 80387 is present in the configuration (according to the state of the ERROR# pin after RESET). If ET is reset, the configuration either contains an 80287 or does not contain a coprocessor. A software test is required to distinguish between these latter two possibilities.

The remaining registers and flags are set as follows:

```
EFLAGS              =00000002H
IP                  =0000FFF0H
CS selector         =000H
DS selector         =0000H
ES selector         =0000H
SS selector         =0000H
FS selector         =0000H
GS selector         =0000H
IDTR:
           base     =0
           limit    =03FFH
```

All registers not mentioned above are undefined.

These settings imply that the processor begins in real-address mode with interrupts disabled.

**Figure 10-1. Contents of EDX after RESET**

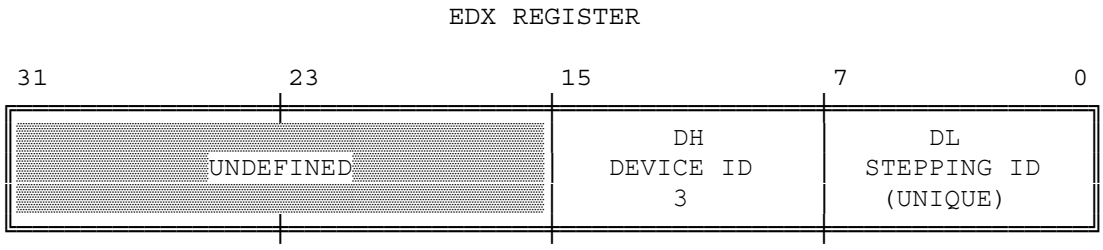
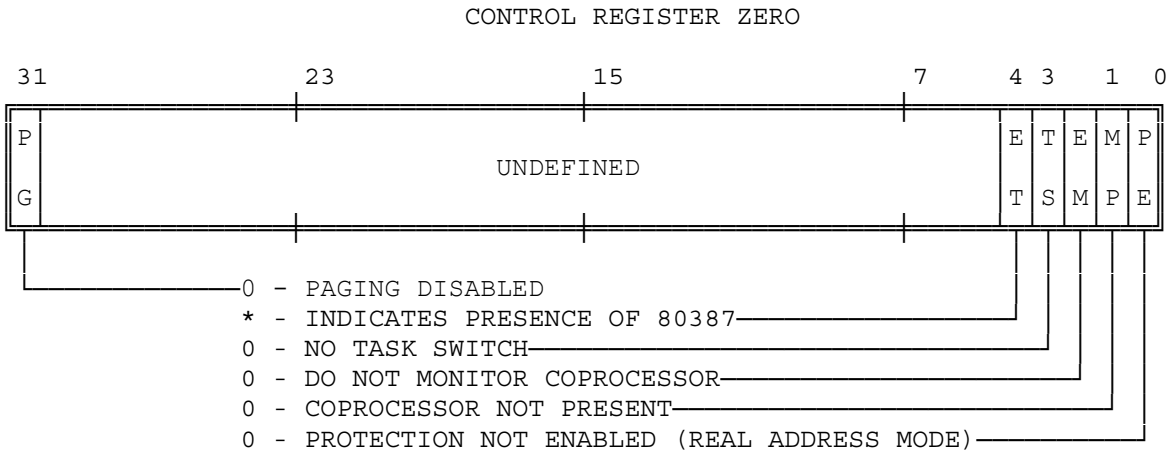



**Figure 10-2. Initial Contents of CR0**




## 10.2 Software Initialization for Real-Address Mode

In real-address mode a few structures must be initialized before a program can take advantage of all the features available in this mode.

### 10.2.1 Stack

No instructions that use the stack can be used until the stack-segment register (SS) has been loaded. SS must point to an area in RAM.

### 10.2.2 Interrupt Table

The initial state of the 80386 leaves interrupts disabled; however, the processor will still attempt to access the interrupt table if an exception or nonmaskable interrupt (NMI) occurs. Initialization software should take one of the following actions:

- Change the limit value in the IDTR to zero. This will cause a shutdown if an exception or nonmaskable interrupt occurs. (Refer to the 80386 Hardware Reference Manual to see how shutdown is signalled externally.)

- Put pointers to valid interrupt handlers in all positions of the interrupt table that might be used by exceptions or interrupts.

- Change the IDTR to point to a valid interrupt table.

### 10.2.3 First Instructions

After RESET, address lines A{31-20} are automatically asserted for instruction fetches. This fact, together with the initial values of CS:IP, causes instruction execution to begin at physical address FFFFFFF0H. Near (intrasegment) forms of control transfer instructions may be used to pass control to other addresses in the upper 64K bytes of the address space. The first far (intersegment) JMP or CALL instruction causes A{31-20} to drop low, and the 80386 continues executing instructions in the lower one megabyte of physical memory. This automatic assertion of address lines A{31-20} allows systems designers to use a ROM at the high end of the address space to initialize the system.

## 10.3 Switching to Protected Mode

Setting the PE bit of the MSW in CR0 causes the 80386 to begin executing in protected mode. The current privilege level (CPL) starts at zero. The segment registers continue to point to the same linear addresses as in real address mode (in real address mode, linear addresses are the same physical addresses).

Immediately after setting the PE flag, the initialization code must flush the processor's instruction prefetch queue by executing a JMP instruction. The 80386 fetches and decodes instructions and addresses before they are used; however, after a change into protected mode, the prefetched instruction information (which pertains to real-address mode) is no longer valid. A JMP forces the processor to discard the invalid information.

## 10.4 Software Initialization for Protected Mode

Most of the initialization needed for protected mode can be done either before or after switching to protected mode. If done in protected mode, however, the initialization procedures must not use protected-mode features that are not yet initialized.

### 10.4.1 Interrupt Descriptor Table

The IDTR may be loaded in either real-address or protected mode. However, the format of the interrupt table for protected mode is different than that for real-address mode. It is not possible to change to protected mode and change interrupt table formats at the same time; therefore, it is inevitable that, if IDTR selects an interrupt table, it will have the wrong format at some time. An interrupt or exception that occurs at this time will have unpredictable results. To avoid this unpredictability, interrupts should remain disabled until interrupt handlers are in place and a valid IDT has been created in protected mode.

### 10.4.2 Stack

The SS register may be loaded in either real-address mode or protected mode. If loaded in real-address mode, SS continues to point to the same linear base-address after the switch to protected mode.

### 10.4.3 Global Descriptor Table

Before any segment register is changed in protected mode, the GDT register must point to a valid GDT. Initialization of the GDT and GDTR may be done in real-address mode. The GDT (as well as LDTs) should reside in RAM, because the processor modifies the accessed bit of descriptors.

### 10.4.4 Page Tables

Page tables and the PDBR in CR3 can be initialized in either real-address mode or in protected mode; however, the paging enabled (PG) bit of CR0 cannot be set until the processor is in protected mode. PG may be set simultaneously with PE, or later. When PG is set, the PDBR in CR3 should already be initialized with a physical address that points to a valid page directory. The initialization procedure should adopt one of the following strategies to ensure consistent addressing before and after paging is enabled:

- The page that is currently being executed should map to the same physical addresses both before and after PG is set.
- A JMP instruction should immediately follow the setting of PG.

### 10.4.5 First Task

The initialization procedure can run awhile in protected mode without initializing the task register; however, before the first task switch, the following conditions must prevail:

- There must be a valid task state segment (TSS) for the new task. The stack pointers in the TSS for privilege levels numerically less than or equal to the initial CPL must point to valid stack segments.
- The task register must point to an area in which to save the current task state. After the first task switch, the information dumped in this area is not needed, and the area can be used for other purposes.

## 10.5 Initialization Example

```
$TITLE ('Initial Task')

    NAME    INIT

init_stack  SEGMENT RW
            DW  20  DUP(?)
tos         LABEL   WORD
init_stack  ENDS

init_data   SEGMENT RW PUBLIC
            DW  20  DUP(?)
init_data   ENDS

init_code   SEGMENT ER PUBLIC

ASSUME      DS:init_data

    nop
    nop
    nop
init_start:
                                        ; set up stack
    mov ax, init_stack
    mov ss, ax
    mov esp, offset tos

    mov a1,1
blink:
    xor a1,1
    out 0e4h,a1
    mov cx,3FFFh
here:
    dec cx
    jnz here

    jmp SHORT blink
```

```
    hlt
init_code   ends

    END init_start, SS:init_stack, DS:init_data

$TITLE('Protected Mode Transition -- 386 initialization')
NAME  RESET

;*****************************************************************
; Upon reset the 386 starts executing at address 0FFFFFFF0H.  The
; upper 12 address bits remain high until a FAR call or jump is
; executed.
;
; Assume the following:
;
;
; -  a short jump at address 0FFFFFFF0H (placed there by the
;    system builder) causes execution to begin at START in segment
;    RESET_CODE.
;
;
; -  segment RESET_CODE is based at physical address 0FFFF0000H,
;    i.e.   at the start of the last  64K in the 4G address space.
;    Note that  this is the base of the CS register at reset.  If
;    you locate ROMcode above  this  address,  you  will  need  to
;    figure out an adjustment factor to address things within this
;    segment.
;
;*****************************************************************
$EJECT ;

; Define addresses to locate GDT and IDT in RAM.
; These addresses are also used in the BLD386 file that defines
; the GDT and IDT. If you change these addresses, make sure you
; change the base addresses specified in the build file.

GDTbase         EQU    00001000H   ; physical address for GDT base
IDTbase         EQU    00000400H   ; physical address for IDT base

PUBLIC     GDT_EPROM
PUBLIC     IDT_EPROM
PUBLIC     START

DUMMY      segment rw      ; ONLY for ASM386 main module stack init
           DW 0
DUMMY   ends

;*****************************************************************
;
; Note: RESET CODE must be USEl6 because the 386 initally executes
;       in real mode.
;

RESET_CODE segment er PUBLIC    USE16

ASSUME DS:nothing, ES:nothing
```

```
;
; 386 Descriptor template

DESC       STRUC
    lim_0_15    DW  0              ; limit bits (0..15)
    bas_0_15    DW  0              ; base bits (0..15)
    bas_16_23   DB  0              ; base bits (16..23)
    access      DB  0              ; access byte
    gran        DB  0              ; granularity byte
    bas_24_31   DB  0              ; base bits (24..31)
DESC       ENDS

; The following is the layout of the real GDT created by BLD386.
; It is located in EPROM and will be copied to RAM.
;
; GDT[O]      ...  NULL
; GDT[1]      ...  Alias for RAM GDT
; GDT[2]      ...  Alias for RAM IDT
; GDT[2]      ...  initial task TSS
; GDT[3]      ...  initial task TSS alias
; GDT[4]      ...  initial task LDT
; GDT[5]      ...  initial task LDT alias

;
; define entries in GDT and IDT.

GDT_ENTRIES    EQU    8
IDT_ENTRIES    EQU    32

; define some constants to index into the real GDT

GDT_ALIAS      EQU    1*SIZE DESC
IDT_ALIAS      EQU    2*SIZE DESC
INIT_TSS       EQU    3*SIZE DESC
INIT_TSS_A     EQU    4*SIZE DESC
INIT_LDT       EQU    5*SIZE DESC
INIT_LDT_A     EQU    6*SIZE DESC

;
; location of alias in INIT_LDT

INIT_LDT_ALIAS    EQU    1*SIZE DESC

;
; access rights byte for DATA and TSS descriptors

DS_ACCESS   EQU   010010010B
TSS_ACCESS  EQU   010001001B


;
; This temporary GDT will be used to set up the real GDT in RAM.

Temp_GDT    LABEL   BYTE        ; tag for begin of scratch GDT

NULL_DES    DESC <>             ; NULL descriptor
```

```
                                ; 32-Gigabyte data segment based at 0
FLAT_DES    DESC <0FFFFH,0,0,92h,0CFh,0>

GDT_eprom     DP    ?           ; Builder places GDT address and limit
                                ; in this 6 byte area.

IDT_eprom     DP    ?           ; Builder places IDT address and limit
                                ; in this 6 byte area.

;
; Prepare operand for loadings GDTR and LDTR.


TGDT_pword     LABEL  PWORD                 ; for temp GDT
        DW     end_Temp_GDT_Temp_GDT -1
        DD     0

GDT_pword      LABEL  PWORD                 ; for GDT in RAM
        DW     GDT_ENTRIES * SIZE DESC -1
        DD     GDTbase

IDT_pword      LABEL   PWORD                ; for IDT in RAM
        DW     IDT_ENTRIES * SIZE DESC -1
        DD     IDTbase


end_Temp_GDT   LABEL   BYTE

;
; Define equates for addressing convenience.

GDT_DES_FLAT        EQU DS:GDT_ALIAS +GDTbase
IDT_DES_FLAT        EQU DS:IDT_ALIAS +GDTbase

INIT_TSS_A_OFFSET   EQU DS:INIT_TSS_A
INIT_TSS_OFFSET     EQU DS:INIT_TSS

INIT_LDT_A_OFFSET   EQU DS:INIT_LDT_A
INIT_LDT_OFFSET     EQU DS:INIT_LDT


; define pointer for first task switch

ENTRY POINTER LABEL DWORD
             DW 0, INIT_TSS

;*****************************************************************
;
;   Jump from reset vector to here.

START:

    CLI                ;disable interrupts
    CLD                ;clear direction flag

    LIDT    NULL_des   ;force shutdown on errors
```

```
;
;   move scratch GDT to RAM at physical 0

    XOR DI,DI
    MOV ES,DI           ;point ES:DI to physical location 0


    MOV SI,OFFSET Temp_GDT
    MOV CX,end_Temp_GDT-Temp_GDT        ;set byte count
    INC CX
;
;   move table

    REP MOVS BYTE PTR ES:[DI],BYTE PTR CS:[SI]

    LGDT    tGDT_pword                  ;load GDTR for Temp. GDT
                                        ;(located at 0)

;   switch to protected mode

    MOV EAX,CR0                         ;get current CRO
    MOV EAX,1                           ;set PE bit
    MOV CRO,EAX                         ;begin protected mode
;
;   clear prefetch queue

    JMP SHORT flush
flush:

; set DS,ES,SS to address flat linear space (0 ... 4GB)

    MOV BX,FLAT_DES-Temp_GDT
    MOV US,BX
    MOV ES,BX
    MOV SS,BX
;
; initialize stack pointer to some (arbitrary) RAM location

    MOV ESP, OFFSET end_Temp_GDT

;
; copy eprom GDT to RAM

    MOV ESI,DWORD PTR GDT_eprom +2 ; get base of eprom GDT
                                   ; (put here by builder).

    MOV EDI,GDTbase                ; point ES:EDI to GDT base in RAM.

    MOV CX,WORD PTR gdt_eprom +0   ; limit of eprom GDT
    INC CX
    SHR CX,1                       ; easier to move words
    CLD
    REP MOVS   WORD PTR ES:[EDI],WORD PTR DS:[ESI]

;
; copy eprom IDT to RAM
;
```

```
    MOV ESI,DWORD PTR IDT_eprom +2 ; get base of eprom IDT
                                   ; (put here by builder)

    MOV EDI,IDTbase                ; point ES:EDI to IDT base in RAM.

    MOV CX,WORD PTR idt_eprom +0   ; limit of eprom IDT
    INC CX
    SHR CX,1
    CLD
    REP MOVS   WORD PTR ES:[EDI],WORD PTR DS:[ESI]

; switch to RAM GDT and IDT
;
    LIDT IDT_pword
    LGDT GDT_pword

;
    MOV BX,GDT_ALIAS               ; point DS to GDT alias
    MOV DS,BX
;
; copy eprom TSS to RAM
;
    MOV BX,INIT_TSS_A              ; INIT TSS A descriptor base
                                   ; has RAM location of INIT TSS.

    MOV ES,BX                      ; ES points to TSS in RAM

    MOV BX,INIT_TSS                ; get inital task selector
    LAR DX,BX                      ; save access byte
    MOV [BX].access,DS_ACCESS      ; set access as data segment
    MOV FS,BX                      ; FS points to eprom TSS

    XOR si,si                      ; FS:si points to eprom TSS
    XOR di,di                      ; ES:di points to RAM TSS

    MOV CX,[BX].lim_0_15           ; get count to move
    INC CX

;
; move INIT_TSS to RAM.

    REP MOVS BYTE PTR ES:[di],BYTE PTR FS:[si]

    MOV [BX].access,DH             ; restore access byte

;
; change base of INIT TSS descriptor to point to RAM.

    MOV AX,INIT_TSS_A_OFFSET.bas_0_15
    MOV INIT_TSS_OFFSET.bas_0_15,AX
    MOV AL,INIT_TSS_A_OFFSET.bas_16_23
    MOV INIT_TSS_OFFSET.bas_16_23,AL
    MOV AL,INIT_TSS_A_OFFSET.bas_24_31
    MOV INIT_TSS_OFFSET.bas_24_31,AL

;
; change INIT TSS A to form a save area for TSS on first task
```

```
; switch. Use RAM at location 0.

    MOV BX,INIT_TSS_A
    MOV WORD PTR [BX].bas_0_15,0
    MOV [BX].bas_16_23,0
    MOV [BX].bas_24_31,0
    MOV [BX].access,TSS_ACCESS
    MOV [BX].gran,O
    LTR BX                          ; defines save area for TSS

;
; copy eprom LDT to RAM

    MOV BX,INIT_LDT_A               ; INIT_LDT_A descriptor has
                                    ; base address in RAM for INIT_LDT.

    MOV ES,BX                       ; ES points LDT location in RAM.

    MOV AH,[BX].bas_24_31
    MOV AL,[BX].bas_16_23
    SHL EAX,16
    MOV AX,[BX].bas_0_15            ; save INIT_LDT base (ram) in EAX

    MOV BX,INIT_LDT                 ; get inital LDT selector
    LAR DX,BX                       ; save access rights
    MOV [BX].access,DS_ACCESS       ; set access as data segment
    MOV FS,BX                       ; FS points to eprom LDT

    XOR si,si                       ; FS:SI points to eprom LDT
    XOR di,di                       ; ES:DI points to RAM LDT

    MOV CX,[BX].lim_0_15            ; get count to move
    INC CX
;
; move initial LDT to RAM

    REP MOVS BYTE PTR ES:[di],BYTE PTR FS:[si]

    MOV [BX].access,DH              ; restore access rights in
                                    ; INIT_LDT descriptor

;
; change base of alias (of INIT_LDT) to point to location in RAM.

    MOV ES:[INIT_LDT_ALIAS].bas_0_15,AX
    SHR EAX,16
    MOV ES:[INIT_LDT_ALIAS].bas_16_23,AL
    MOV ES:[INIT_LDT_ALIAS].bas_24_31,AH

;
; now set the base value in INIT_LDT descriptor

    MOV AX,INIT_LDT_A_OFFSET.bas_0_15
    MOV INIT_LDT_OFFSET.bas_0_15,AX
    MOV AL,INIT_LDT_A_OFFSET.bas_16_23
    MOV INIT_LDT_OFFSET.bas_16_23,AL
    MOV AL,INIT_LDT_A_OFFSET.bas_24_31
```

```
    MOV INIT_LDT_OFFSET.bas_24_31,AL

;
; Now GDT, IDT, initial TSS and initial LDT are all set up.
;
; Start the first task!
'
   JMP ENTRY_POINTER

RESET_CODE ends
   END START, SS:DUMMY,DS:DUMMY
```

## 10.6 TLB Testing

The 80386 provides a mechanism for testing the Translation Lookaside Buffer (TLB), the cache used for translating linear addresses to physical addresses. Although failure of the TLB hardware is extremely unlikely, users may wish to include TLB confidence tests among other power-up confidence tests for the 80386.

---

NOTE
This TLB testing mechanism is unique to the 80386 and may not be continued in the same way in future processors. Sortware that uses this mechanism may be incompatible with future processors.

---

When testing the TLB it is recommended that paging be turned off (PG=0 in CR0) to avoid interference with the test data being written to the TLB.

### 10.6.1 Structure of the TLB

The TLB is a four-way set-associative memory. Figure 10-3 illustrates the structure of the TLB. There are four sets of eight entries each. Each entry consists of a tag and data. Tags are 24-bits wide. They contain the high-order 20 bits of the linear address, the valid bit, and three attribute bits. The data portion of each entry contains the high-order 20 bits of the physical address.

### 10.6.2 Test Registers

Two test registers, shown in Figure 10-4, are provided for the purpose of testing. TR6 is the test command register, and TR7 is the test data register. These registers are accessed by variants of the MOV instruction. A test register may be either the source operand or destination operand. The MOV instructions are defined in both real-address mode and protected mode. The test registers are privileged resources; in protected mode, the MOV instructions that access them can only be executed at privilege level 0. An attempt to read or write the test registers when executing at any other privilege level causes a general protection exception.

The test command register (TR6) contains a command and an address tag to use in performing the command:

C — This is the command bit. There are two TLB testing commands: write entries into the TLB, and perform TLB lookups. To cause an immediate write into the TLB entry, move a doubleword into TR6 that contains a 0 in this bit. To cause an immediate TLB lookup, move a doubleword into TR6 that contains a 1 in this bit.

Linear Address — On a TLB write, a TLB entry is allocated to this linear address; the rest of that TLB entry is set per the value of TR7 and the value just written into TR6. On a TLB lookup, the TLB is interrogated per this value; if one and only one TLB entry matches, the rest of the fields of TR6 and TR7 are set from the matching TLB entry.

V — The valid bit for this TLB entry. The TLB uses the valid bit to identify entries that contain valid data. Entries of the TLB that have not been assigned values have zero in the valid bit. All valid bits can be cleared by writing to CR3.

D, D# — The dirty bit (and its complement) for/from the TLB entry.

U, U# — The U/S bit (and its complement) for/from the TLB entry.

W, W# — The R/W bit (and its complement) for/from the TLB entry.

The meaning of these pairs of bits is given by Table 10-1, where X represents D, U, or W.

The test data register (TR7) holds data read from or data to be written to the TLB.

Physical Address — This is the data field of the TLB. On a write to the TLB, the TLB entry allocated to the linear address in TR6 is set to this value. On a TLB lookup, if HT is set, the data field (physical address) from the TLB is read out to this field. If HT is not set, this field is undefined.

HT — For a TLB lookup, the HT bit indicates whether the lookup was a hit (HT ← 1) or a miss (HT ← 0). For a TLB write, HT must be set to 1.

REP — For a TLB write, selects which of four associative blocks of the TLB is to be written. For a TLB read, if HT is set, REP reports in which of the four associative blocks the tag was found; if HT is not set, REP is undefined.

**Table 10-1. Meaning of D, U, and W Bit Pairs**

| X | X# | Effect during TLB Lookup | Value of bit X after TLB Write |
|---|---|---|---|
| 0 | 0 | (undefined) | (undefined) |
| 0 | 1 | Match if X=0 | Bit X becomes 0 |
| 1 | 0 | Match if X=1 | Bit X becomes 1 |
| 1 | 1 | (undefined) | (undefined) |

**Figure 10-3. TLB Structure**

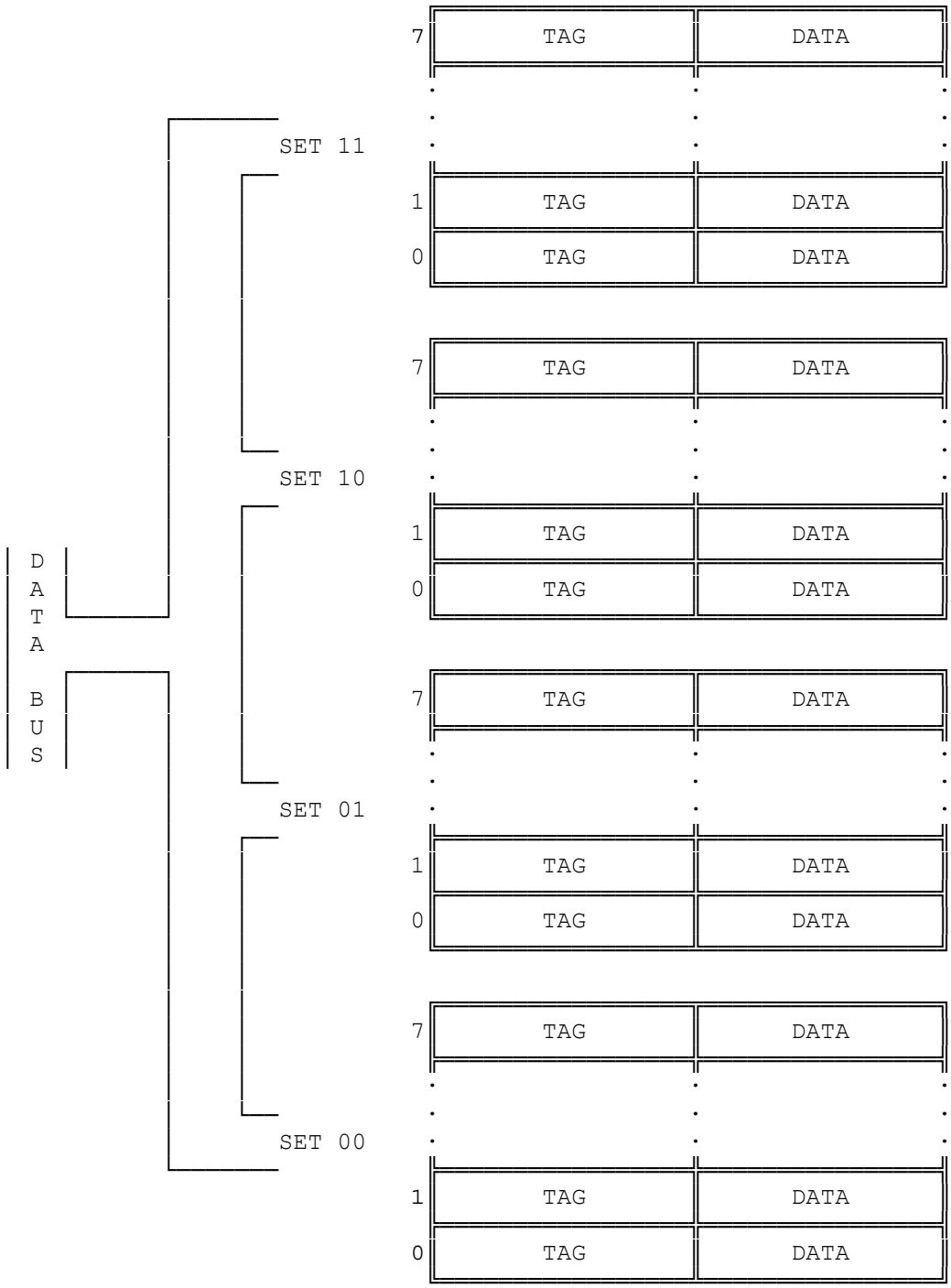
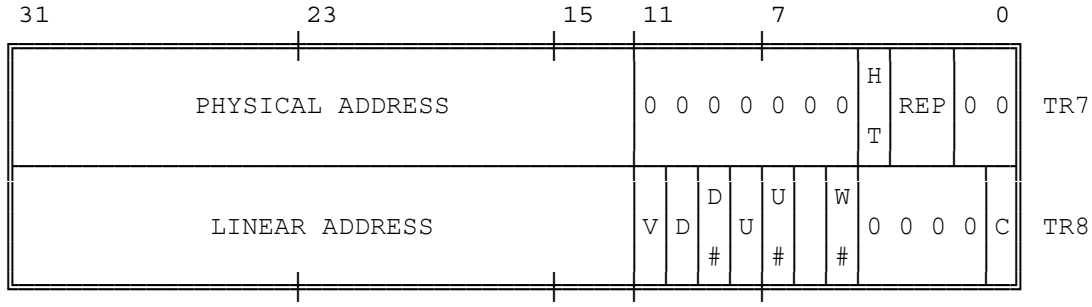


**Figure 10-4. Test Registers**




NOTE: 0 INDICATES INTEL RESERVED. NO NOT DEFINE

### 10.6.3 Test Operations

To write a TLB entry:

1. Move a doubleword to TR7 that contains the desired physical address, HT, and REP values. HT must contain 1. REP must point to the associative block in which to place the entry.

2. Move a doubleword to TR6 that contains the appropriate linear address, and values for V, D, U, and W. Be sure C=0 for "write" command.

Be careful not to write duplicate tags; the results of doing so are undefined.

To look up (read) a TLB entry:

1. Move a doubleword to TR6 that contains the appropriate linear address and attributes. Be sure C=1 for "lookup" command.

2. Store TR7. If the HT bit in TR7 indicates a hit, then the other values reveal the TLB contents. If HT indicates a miss, then the other values in TR7 are indeterminate.

For the purposes of testing, the V bit functions as another bit of addresss. The V bit for a lookup request should usually be set, so that uninitialized tags do not match. Lookups with V=0 are unpredictable if any tags are uninitialized.

# Chapter 11 Coprocessing and Multiprocessing

The 80386 has two levels of support for multiple parallel processing units:

- A highly specialized interface for very closely coupled processors of a type known as coprocessors.
- A more general interface for more loosely coupled processors of unspecified type.

## 11.1 Coprocessing

The components of the coprocessor interface include:

- ET bit of control register zero (CR0)
- The EM, and MP bits of CR0
- The ESC instructions
- The WAIT instruction
- The TS bit of CR0
- Exceptions

### 11.1.1 Coprocessor Identification

The 80386 is designed to operate with either an 80287 or 80387 math coprocessor. The ET bit of CR0 indicates which type of coprocessor is present. ET is set automatically by the 80386 after RESET according to the level detected on the ERROR# input. If desired, ET may also be set or reset by loading CR0 with a MOV instruction. If ET is set, the 80386 uses the 32-bit protocol of the 80387; if reset, the 80386 uses the 16-bit protocol of the 80287.

### 11.1.2 ESC and WAIT Instructions

The 80386 interprets the pattern 11011B in the first five bits of an instruction as an opcode intended for a coprocessor. Instructions thus marked are called ESCAPE or ESC instructions. The CPU performs the following functions upon encountering an ESC instruction before sending the instruction to the coprocessor:

- Tests the emulation mode (EM) flag to determine whether coprocessor functions are being emulated by software.
- Tests the TS flag to determine whether there has been a context change since the last ESC instruction.

- For some ESC instructions, tests the ERROR# pin to determine whether the coprocessor detected an error in the previous ESC instruction.

The WAIT instruction is not an ESC instruction, but WAIT causes the CPU to perform some of the same tests that it performs upon encountering an ESC instruction. The processor performs the following actions for a WAIT instruction:

- Waits until the coprocessor no longer asserts the BUSY# pin.
- Tests the ERROR# pin (after BUSY# goes inactive). If ERROR# is active, the 80386 signals exception 16, which indicates that the coprocessor encountered an error in the previous ESC instruction.
- WAIT can therefore be used to cause exception 16 if an error is pending from a previous ESC instruction. Note that, if no coprocessor is present, the ERROR# and BUSY# pins should be tied inactive to prevent WAIT from waiting forever or causing spurious exceptions.

### 11.1.3 EM and MP Flags

The EM and MP flags of CR0 control how the processor reacts to coprocessor instructions.

The EM bit indicates whether coprocessor functions are to be emulated. If the processor finds EM set when executing an ESC instruction, it signals exception 7, giving the exception handler an opportunity to emulate the ESC instruction.

The MP (monitor coprocessor) bit indicates whether a coprocessor is actually attached. The MP flag controls the function of the WAIT instruction. If, when executing a WAIT instruction, the CPU finds MP set, then it tests the TS flag; it does not otherwise test TS during a WAIT instruction. If it finds TS set under these conditions, the CPU signals exception 7.

The EM and MP flags can be changed with the aid of a MOV instruction using CR0 as the destination operand and read with the aid of a MOV instruction with CR0 as the source operand. These forms of the MOV instruction can be executed only at privilege level zero.

### 11.1.4 The Task-Switched Flag

The TS bit of CR0 helps to determine when the context of the coprocessor does not match that of the task being executed by the 80386 CPU. The 80386 sets TS each time it performs a task switch (whether triggered by software or by hardware interrupt). If, when interpreting one of the ESC instructions, the CPU finds TS already set, it causes exception 7. The WAIT instruction also causes exception 7 if both TS and MP are set. Operating systems can use this exception to switch the context of the coprocessor to correspond to the current task. Refer to the 80386 System Software Writer's Guide for an example.

The CLTS instruction (legal only at privilege level zero) resets the TS flag.

### 11.1.5 Coprocessor Exceptions

Three exceptions aid in interfacing to a coprocessor: interrupt 7 (coprocessor not available), interrupt 9 (coprocessor segment overrun), and interrupt 16 (coprocessor error).

#### 11.1.5.1 Interrupt 7 —— Coprocessor Not Available

This exception occurs in either of two conditions:

1. The CPU encounters an ESC instruction and EM is set. In this case, the exception handler should emulate the instruction that caused the exception. TS may also be set.

2. The CPU encounters either the WAIT instruction or an ESC instruction when both MP and TS are set. In this case, the exception handler should update the state of the coprocessor, if necessary.

#### 11.1.5.2 Interrupt 9 —— Coprocessor Segment Overrun

This exception occurs in protected mode under the following conditions:

- An operand of a coprocessor instruction wraps around an addressing limit (0FFFFH for small segments, 0FFFFFFFFH for big segments, zero for expand-down segments). An operand may wrap around an addressing limit when the segment limit is near an addressing limit and the operand is near the largest valid address in the segment. Because of the wrap-around, the beginning and ending addresses of such an operand will be near opposite ends of the segment.

- Both the first byte and the last byte of the operand (considering wrap-around) are at addresses located in the segment and in present and accessible pages.

- The operand spans inaccessible addresses. There are two ways that such an operand may also span inaccessible addresses:

    1. The segment limit is not equal to the addressing limit (e.g., addressing limit is FFFFH and segment limit is FFFDH); therefore, the operand will span addresses that are not within the segment (e.g., an 8-byte operand that starts at valid offset FFFC will span addresses FFFC-FFFF and 0000-0003; however, addresses FFFE and FFFF are not valid, because they exceed the limit);

    2. The operand begins and ends in present and accessible pages but intermediate bytes of the operand fall either in a not-present page or in a page to which the current procedure does not have access rights.

The address of the failing numerics instruction and data operand may be lost; an FSTENV does not return reliable addresses. As with the 80286/80287, the segment overrun exception should be handled by executing an FNINIT instruction (i.e., an FINIT without a preceding WAIT). The return address on the stack does not necessarily point to the failing instruction nor to the following instruction. The failing numerics instruction is not restartable.

Case 2 can be avoided by either aligning all segments on page boundaries or by not starting them within 108 bytes of the start or end of a page. (The maximum size of a coprocessor operand is 108 bytes.) Case 1 can be avoided by making sure that the gap between the last valid offset and the first valid offset of a segment is either no less than 108 bytes or is zero (i.e., the segment is of full size). If neither software system design constraint is acceptable, the exception handler should execute FNINIT and should probably terminate the task.

#### 11.1.5.3 Interrupt 16 —— Coprocessor Error

The numerics coprocessors can detect six different exception conditions during instruction execution. If the detected exception is not masked by a bit in the control word, the coprocessor communicates the fact that an error occurred to the CPU by a signal at the ERROR# pin. The CPU causes interrupt 16 the next time it checks the ERROR# pin, which is only at the beginning of a subsequent WAIT or certain ESC instructions. If the exception is masked, the numerics coprocessor handles the exception according to on-board logic; it does not assert the ERROR# pin in this case.

## 11.2 General Multiprocessing

The components of the general multiprocessing interface include:

- The LOCK# signal
- The LOCK instruction prefix, which gives programmed control of the LOCK# signal.
- Automatic assertion of the LOCK# signal with implicit memory updates by the processor

### 11.2.1 LOCK and the LOCK# Signal

The LOCK instruction prefix and its corresponding output signal LOCK# can be used to prevent other bus masters from interrupting a data movement operation. LOCK may only be used with the following 80386 instructions when they modify memory. An undefined-opcode exception results from using LOCK before any instruction other than:

- Bit test and change: BTS, BTR, BTC.
- Exchange: XCHG.
- Two-operand arithmetic and logical: ADD, ADC, SUB, SBB, AND, OR, XOR.
- One-operand arithmetic and logical: INC, DEC, NOT, and NEG.

A locked instruction is only guaranteed to lock the area of memory defined by the destination operand, but it may lock a larger memory area. For example, typical 8086 and 80286 configurations lock the entire physical memory space. The area of memory defined by the destination operand is guaranteed to be locked against access by a processor executing a locked instruction on exactly the same memory area, i.e., an operand with identical starting address and identical length.

The integrity of the lock is not affected by the alignment of the memory field. The LOCK signal is asserted for as many bus cycles as necessary to update the entire operand.

### 11.2.2 Automatic Locking

In several instances, the processor itself initiates activity on the data bus. To help ensure that such activities function correctly in multiprocessor configurations, the processor automatically asserts the LOCK# signal. These instances include:

- Acknowledging interrupts.

  After an interrupt request, the interrupt controller uses the data bus to send the interrupt ID of the interrupt source to the CPU. The CPU asserts LOCK# to ensure that no other data appears on the data bus during this time.

- Setting busy bit of TSS descriptor.

  The processor tests and sets the busy-bit in the type field of the TSS descriptor when switching to a task. To ensure that two different processors cannot simultaneously switch to the same task, the processor asserts LOCK# while testing and setting this bit.

- Loading of descriptors.

  While copying the contents of a descriptor from a descriptor table into a segment register, the processor asserts LOCK# so that the descriptor cannot be modified by another processor while it is being loaded. For this action to be effective, operating-system procedures that update descriptors should adhere to the following steps:

  — Use a locked update to the access-rights byte to mark the descriptor not-present.

  — Update the fields of the descriptor. (This may require several memory accesses; therefore, LOCK cannot be used.)

  — Use a locked update to the access-rights byte to mark the descriptor present again.

- Updating page-table A and D bits.

  The processor exerts LOCK# while updating the A (accessed) and D (dirty) bits of page-table entries. Also the processor bypasses the page-table cache and directly updates these bits in memory.

- Executing XCHG instruction.

  The 80386 always asserts LOCK during an XCHG instruction that references memory (even if the LOCK prefix is not used).

### 11.2.3 Cache Considerations

Systems programmers must take care when updating shared data that may also be stored in on-chip registers and caches. With the 80386, such shared data includes:

- Descriptors, which may be held in segment registers.

  A change to a descriptor that is shared among processors should be broadcast to all processors. Segment registers are effectively "descriptor caches". A change to a descriptor will not be utilized by another processor if that processor already has a copy of the old version of the descriptor in a segment register.

- Page tables, which may be held in the page-table cache.

  A change to a page table that is shared among processors should be broadcast to all processors, so that others can flush their page-table caches and reload them with up-to-date page tables from memory.

Systems designers can employ an interprocessor interrupt to handle the above cases. When one processor changes data that may be cached by other processors, it can send an interrupt signal to all other processors that may be affected by the change. If the interrupt is serviced by an interrupt task, the task switch automatically flushes the segment registers. The task switch also flushes the page-table cache if the PDBR (the contents of CR3) of the interrupt task is different from the PDBR of every other task.

In multiprocessor systems that need a cacheability signal from the CPU, it is recommended that physical address pin A31 be used to indicate cacheability. Such a system can then possess up to 2 Gbytes of physical memory. The virtual address range available to the programmer is not affected by this convention.

# Chapter 12 Debugging

The 80386 brings to Intel's line of microprocessors significant advances in debugging power. The single-step exception and breakpoint exception of previous processors are still available in the 80386, but the principal debugging support takes the form of debug registers. The debug registers support both instruction breakpoints and data breakpoints. Data breakpoints are an important innovation that can save hours of debugging time by pinpointing, for example, exactly when a data structure is being overwritten. The breakpoint registers also eliminate the complexities associated with writing a breakpoint instruction into a code segment (requires a data-segment alias for the code segment) or a code segment shared by multiple tasks (the breakpoint exception can occur in the context of any of the tasks). Breakpoints can even be set in code contained in ROM.

## 12.1 Debugging Features of the Architecture

The features of the 80386 architecture that support debugging include:

Reserved debug interrupt vector

Permits processor to automatically invoke a debugger task or procedure when an event occurs that is of interest to the debugger.

Four debug address registers

Permit programmers to specify up to four addresses that the CPU will automatically monitor.

Debug control register

Allows programmers to selectively enable various debug conditions associated with the four debug addresses.

Debug status register

Helps debugger identify condition that caused debug exception.

Trap bit of TSS (T-bit)

Permits monitoring of task switches.

Resume flag (RF) of flags register

Allows an instruction to be restarted after a debug exception without immediately causing another debug exception due to the same condition.

Single-step flag (TF)

Allows complete monitoring of program flow by specifying whether the CPU should cause a debug exception with the execution of every instruction.

Breakpoint instruction

Permits debugger intervention at any point in program execution and aids debugging of debugger programs.

Reserved interrupt vector for breakpoint exception

Permits processor to automatically invoke a handler task or procedure upon encountering a breakpoint instruction.

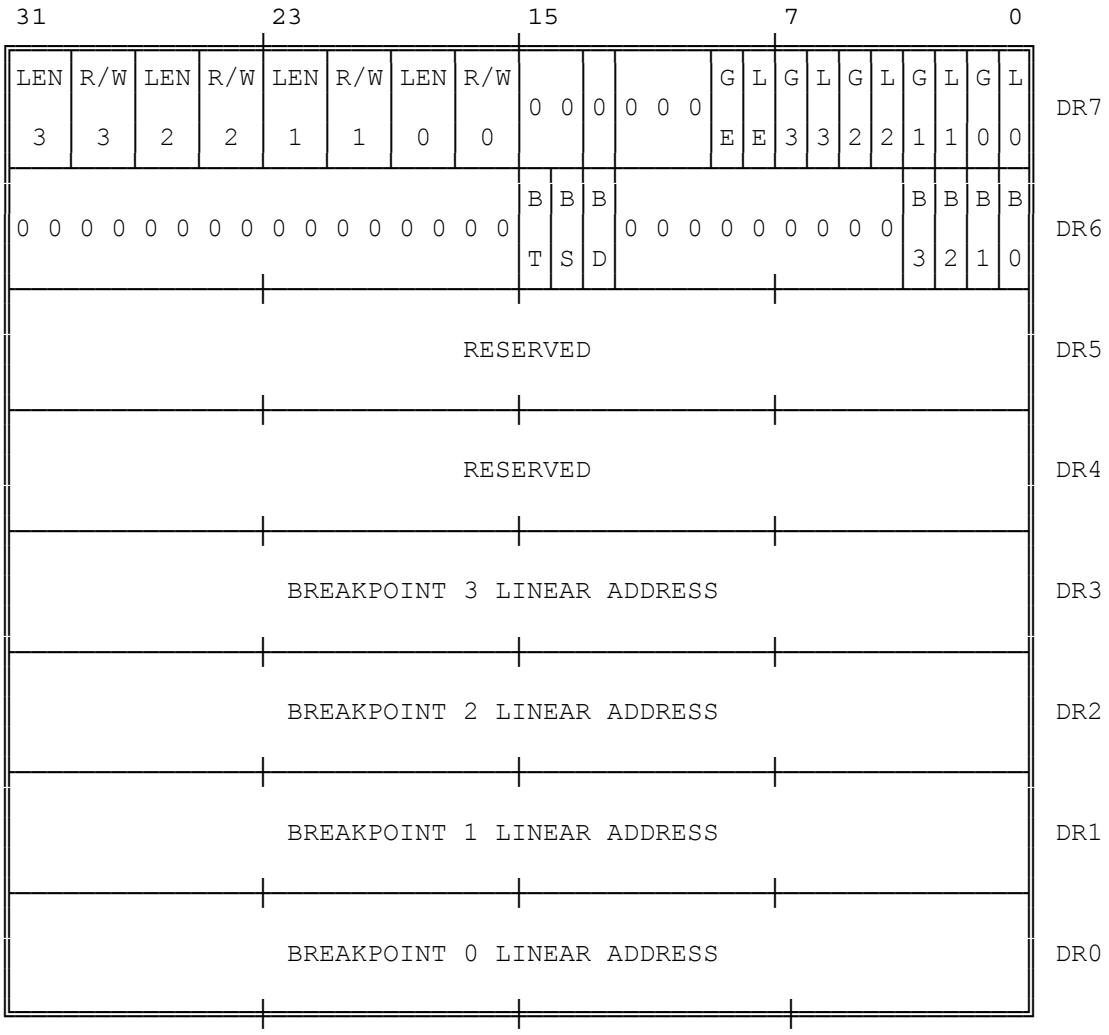
These features make it possible to invoke a debugger that is either a separate task or a procedure in the context of the current task. The debugger can be invoked under any of the following kinds of conditions:

- Task switch to a specific task.
- Execution of the breakpoint instruction.
- Execution of every instruction.
- Execution of any instruction at a given address.
- Read or write of a byte, word, or doubleword at any specified address.
- Write to a byte, word, or doubleword at any specified address.
- Attempt to change a debug register.

## 12.2 Debug Registers

Six 80386 registers are used to control debug features. These registers are accessed by variants of the MOV instruction. A debug register may be either the source operand or destination operand. The debug registers are privileged resources; the MOV instructions that access them can only be executed at privilege level zero. An attempt to read or write the debug registers when executing at any other privilege level causes a general protection exception. Figure 12-1 shows the format of the debug registers.

**Figure 12-1. Debug Registers**




NOTE
0 MEANS INTEL RESERVED. DO NOT DEFINE.

## 12.2.1 Debug Address Registers (DR0-DR3)

Each of these registers contains the linear address associated with one of four breakpoint conditions. Each breakpoint condition is further defined by bits in DR7.

The debug address registers are effective whether or not paging is enabled. The addresses in these registers are linear addresses. If paging is enabled, the linear addresses are translated into physical addresses by the processor's paging mechanism (as explained in Chapter 5). If paging is not enabled, these linear addresses are the same as physical addresses.

Note that when paging is enabled, different tasks may have different linear-to-physical address mappings. When this is the case, an address in a debug address register may be relevant to one task but not to another. For this reason the 80386 has both global and local enable bits in DR7. These bits indicate whether a given debug address has a global (all tasks) or local (current task only) relevance.

### 12.2.2 Debug Control Register (DR7)

The debug control register shown in Figure 12-1 both helps to define the debug conditions and selectively enables and disables those conditions.

For each address in registers DR0-DR3, the corresponding fields R/W0 through R/W3 specify the type of action that should cause a breakpoint. The processor interprets these bits as follows:

00 —— Break on instruction execution only
01 —— Break on data writes only
10 —— undefined
11 —— Break on data reads or writes but not instruction fetches

Fields LEN0 through LEN3 specify the length of data item to be monitored. A length of 1, 2, or 4 bytes may be specified. The values of the length fields are interpreted as follows:

00 —— one-byte length
01 —— two-byte length
10 —— undefined
11 —— four-byte length

If RWn is 00 (instruction execution), then LENn should also be 00. Any other length is undefined.

The low-order eight bits of DR7 (L0 through L3 and G0 through G3) selectively enable the four address breakpoint conditions. There are two levels of enabling: the local (L0 through L3) and global (G0 through G3) levels. The local enable bits are automatically reset by the processor at every task switch to avoid unwanted breakpoint conditions in the new task. The global enable bits are not reset by a task switch; therefore, they can be used for conditions that are global to all tasks.

The LE and GE bits control the "exact data breakpoint match" feature of the processor. If either LE or GE is set, the processor slows execution so that data breakpoints are reported on the instruction that causes them. It is recommended that one of these bits be set whenever data breakpoints are armed. The processor clears LE at a task switch but does not clear GE.

### 12.2.3 Debug Status Register (DR6)

The debug status register shown in Figure 12-1 permits the debugger to determine which debug conditions have occurred.

When the processor detects an enabled debug exception, it sets the low-order bits of this register (B0 thru B3) before entering the debug exception handler. Bn is set if the condition described by DRn, LENn, and R/Wn occurs. (Note that the processor sets Bn regardless of whether Gn or Ln is set. If more than one breakpoint condition occurs at one time and if the breakpoint trap occurs due to an enabled condition other than n, Bn may be set, even though neither Gn nor Ln is set.)

The BT bit is associated with the T-bit (debug trap bit) of the TSS (refer to 7 for the location of the T-bit). The processor sets the BT bit before entering the debug handler if a task switch has occurred and the T-bit of the new TSS is set. There is no corresponding bit in DR7 that enables and disables this trap; the T-bit of the TSS is the sole enabling bit.

The BS bit is associated with the TF (trap flag) bit of the EFLAGS register. The BS bit is set if the debug handler is entered due to the occurrence of a single-step exception. The single-step trap is the highest-priority debug exception; therefore, when BS is set, any of the other debug status bits may also be set.

The BD bit is set if the next instruction will read or write one of the eight debug registers and ICE-386 is also using the debug registers at the same time.

Note that the bits of DR6 are never cleared by the processor. To avoid any confusion in identifying the next debug exception, the debug handler should move zeros to DR6 immediately before returning.

### 12.2.4 Breakpoint Field Recognition

The linear address and LEN field for each of the four breakpoint conditions define a range of sequential byte addresses for a data breakpoint. The LEN field permits specification of a one-, two-, or four-byte field. Two-byte fields must be aligned on word boundaries (addresses that are multiples of two) and four-byte fields must be aligned on doubleword boundaries (addresses that are multiples of four). These requirements are enforced by the processor; it uses the LEN bits to mask the low-order bits of the addresses in the debug address registers. Improperly aligned code or data breakpoint addresses will not yield the expected results.

A data read or write breakpoint is triggered if any of the bytes participating in a memory access is within the field defined by a breakpoint address register and the corresponding LEN field. Table 12-1 gives some examples of breakpoint fields with memory references that both do and do not cause traps.

To set a data breakpoint for a misaligned field longer than one byte, it may be desirable to put two sets of entries in the breakpoint register such that each entry is properly aligned and the two entries together span the length of the field.

Instruction breakpoint addresses must have a length specification of one byte (LEN = 00); other values are undefined. The processor recognizes an instruction breakpoint address only when it points to the first byte of an

instruction. If the instruction has any prefixes, the breakpoint address must point to the first prefix.

**Table 12-1. Breakpoint Field Recognition Examples**

| | | Address (hex) | Length |
|---|---|---|---|
| Register Contents | DR0 | 0A0001 | 1 (LEN0 = 00) |
| | DR1 | 0A0002 | 1 (LEN1 = 00) |
| | DR2 | 0B0002 | 2 (LEN2 = 01) |
| | DR3 | 0C0000 | 4 (LEN3 = 11) |
| Some Examples of Memory References That Cause Traps | | 0A0001 | 1 |
| | | 0A0002 | 1 |
| | | 0A0001 | 2 |
| | | 0A0002 | 2 |
| | | 0B0002 | 2 |
| | | 0B0001 | 4 |
| | | 0C0000 | 4 |
| | | 0C0001 | 2 |
| | | 0C0003 | 1 |
| Some Examples of Memory References That Don't Cause Traps | | 0A0000 | 1 |
| | | 0A0003 | 4 |
| | | 0B0000 | 2 |
| | | 0C0004 | 4 |

## 12.3 Debug Exceptions

Two of the interrupt vectors of the 80386 are reserved for exceptions that relate to debugging. Interrupt 1 is the primary means of invoking debuggers designed expressly for the 80386; interrupt 3 is intended for debugging debuggers and for compatibility with prior processors in Intel's 8086 processor family.

### 12.3.1 Interrupt 1 —— Debug Exceptions

The handler for this exception is usually a debugger or part of a debugging system. The processor causes interrupt 1 for any of several conditions. The debugger can check flags in DR6 and DR7 to determine what condition caused the exception and what other conditions might be in effect at the same time. Table 12-2 associates with each breakpoint condition the combination of bits that indicate when that condition has caused the debug exception.

Instruction address breakpoint conditions are faults, while other debug conditions are traps. The debug exception may report either or both at one time. The following paragraphs present details for each class of debug exception.

**Table 12-2. Debug Exception Conditions**

| Flags to Test | Condition |
|---|---|
| BS=1 | Single-step trap |
| B0=1 AND (GE0=1 OR LE0=1) | Breakpoint DR0, LEN0, R/W0 |
| B1=1 AND (GE1=1 OR LE1=1) | Breakpoint DR1, LEN1, R/W1 |
| B2=1 AND (GE2=1 OR LE2=1) | Breakpoint DR2, LEN2, R/W2 |
| B3=1 AND (GE3=1 OR LE3=1) | Breakpoint DR3, LEN3, R/W3 |
| BD=1 | Debug registers not available; in use by ICE-386. |
| BT=1 | Task switch |

#### 12.3.1.1 Instruction Addrees Breakpoint

The processor reports an instruction-address breakpoint before it executes the instruction that begins at the given address; i.e., an instruction-address breakpoint exception is a fault.

The RF (restart flag) permits the debug handler to retry instructions that cause other kinds of faults in addition to debug faults. When it detects a fault, the processor automatically sets RF in the flags image that it pushes onto the stack. (It does not, however, set RF for traps and aborts.)

When RF is set, it causes any debug fault to be ignored during the next instruction. (Note, however, that RF does not cause breakpoint traps to be ignored, nor other kinds of faults.)

The processor automatically clears RF at the successful completion of every instruction except after the IRET instruction, after the POPF instruction, and after a JMP, CALL, or INT instruction that causes a task switch. These instructions set RF to the value specified by the memory image of the EFLAGS register.

The processor automatically sets RF in the EFLAGS image on the stack before entry into any fault handler. Upon entry into the fault handler for instruction address breakpoints, for example, RF is set in the EFLAGS image on the stack; therefore, the IRET instruction at the end of the handler will set RF in the EFLAGS register, and execution will resume at the breakpoint address without generating another breakpoint fault at the same address.

If, after a debug fault, RF is set and the debug handler retries the faulting instruction, it is possible that retrying the instruction will raise other faults. The retry of the instruction after these faults will also be done with RF=1, with the result that debug faults continue to be ignored. The processor clears RF only after successful completion of the instruction.

Real-mode debuggers can control the RF flag by using a 32-bit IRET. A 16-bit IRET instruction does not affect the RF bit (which is in the high-order 16 bits of EFLAGS). To use a 32-bit IRET, the debugger must rearrange the stack so that it holds appropriate values for the 32-bit EIP, CS, and EFLAGS (with RF set in the EFLAGS image). Then executing an IRET with an operand size prefix causes a 32-bit return, popping the RF flag into EFLAGS.

#### 12.3.1.2 Data Address Breakpoint

A data-address breakpoint exception is a trap; i.e., the processor reports a data-address breakpoint after executing the instruction that accesses the given memory item.

When using data breakpoints it is recommended that either the LE or GE bit of DR7 be set also. If either LE or GE is set, any data breakpoint trap is reported exactly after completion of the instruction that accessed the specified memory item. This exact reporting is accomplished by forcing the 80386 execution unit to wait for completion of data operand transfers before beginning execution of the next instruction. If neither GE nor LE is set, data breakpoints may not be reported until one instruction after the data is accessed or may not be reported at all. This is due to the fact that, normally, instruction execution is overlapped with memory transfers to such a degree that execution of the next instruction may begin before memory transfers for the prior instruction are completed.

If a debugger needs to preserve the contents of a write breakpoint location, it should save the original contents before setting a write breakpoint. Because data breakpoints are traps, a write into a breakpoint location will complete before the trap condition is reported. The handler can report the saved value after the breakpoint is triggered. The data in the debug registers can be used to address the new value stored by the instruction that triggered the breakpoint.

#### 12.3.1.3 General Detect Fault

This exception occurs when an attempt is made to use the debug registers at the same time that ICE-386 is using them. This additional protection feature is provided to guarantee that ICE-386 can have full control over the debug-register resources when required. ICE-386 uses the debug-registers; therefore, a software debugger that also uses these registers cannot run while ICE-386 is in use. The exception handler can detect this condition by examining the BD bit of DR6.

#### 12.3.1.4 Single-Step Trap

This debug condition occurs at the end of an instruction if the trap flag (TF) of the flags register held the value one at the beginning of that instruction. Note that the exception does not occur at the end of an instruction that sets TF. For example, if POPF is used to set TF, a single-step trap does not occur until after the instruction that follows POPF.

The processor clears the TF bit before invoking the handler. If TF=1 in the flags image of a TSS at the time of a task switch, the exception occurs after the first instruction is executed in the new task.

The single-step flag is normally not cleared by privilege changes inside a task. INT instructions, however, do clear TF. Therefore, software

debuggers that single-step code must recognize and emulate INT n or INTO rather than executing them directly.

To maintain protection, system software should check the current execution privilege level after any single step interrupt to see whether single stepping should continue at the current privilege level.

The interrupt priorities in hardware guarantee that if an external interrupt occurs, single stepping stops. When both an external interrupt and a single step interrupt occur together, the single step interrupt is processed first. This clears the TF bit. After saving the return address or switching tasks, the external interrupt input is examined before the first instruction of the single step handler executes. If the external interrupt is still pending, it is then serviced. The external interrupt handler is not single-stepped. To single step an interrupt handler, just single step an INT n instruction that refers to the interrupt handler.

#### 12.3.1.5 Task Switch Breakpoint

The debug exception also occurs after a switch to an 80386 task if the T-bit of the new TSS is set. The exception occurs after control has passed to the new task, but before the first instruction of that task is executed. The exception handler can detect this condition by examining the BT bit of the debug status register DR6.

Note that if the debug exception handler is a task, the T-bit of its TSS should not be set. Failure to observe this rule will cause the processor to enter an infinite loop.

### 12.3.2 Interrupt 3 —— Breakpoint Exception

This exception is caused by execution of the breakpoint instruction INT 3. Typically, a debugger prepares a breakpoint by substituting the opcode of the one-byte breakpoint instruction in place of the first opcode byte of the instruction to be trapped. When execution of the INT 3 instruction causes the exception handler to be invoked, the saved value of ES:EIP points to the byte following the INT 3 instruction.

With prior generations of processors, this feature is used extensively for trapping execution of specific instructions. With the 80386, the needs formerly filled by this feature are more conveniently solved via the debug registers and interrupt 1. However, the breakpoint exception is still useful for debugging debuggers, because the breakpoint exception can vector to a different exception handler than that used by the debugger. The breakpoint exception can also be useful when it is necessary to set a greater number of breakpoints than permitted by the debug registers.

# PART III COMPATIBILITY

# Chapter 13 Executing 80286 Protected-Mode Code

## 13.1 80286 Code Executes as a Subset of the 80386

In general, programs designed for execution in protected mode on an 80286 execute without modification on the 80386, because the features of the 80286 are a subset of those of the 80386.

All the descriptors used by the 80286 are supported by the 80386 as long as the Intel-reserved word (last word) of the 80286 descriptor is zero.

The descriptors for data segments, executable segments, local descriptor tables, and task gates are common to both the 80286 and the 80386. Other 80286 descriptors——TSS segment, call gate, interrupt gate, and trap gate——are supported by the 80386. The 80386 also has new versions of descriptors for TSS segment, call gate, interrupt gate, and trap gate that support the 32-bit nature of the 80386. Both sets of descriptors can be used simultaneously in the same system.

For those descriptors that are common to both the 80286 and the 80386, the presence of zeros in the final word causes the 80386 to interpret these descriptors exactly as 80286 does; for example:

| | |
|---|---|
| Base Address | The high-order eight bits of the 32-bit base address are zero, limiting base addresses to 24 bits. |
| Limit | The high-order four bits of the limit field are zero, restricting the value of the limit field to 64K. |
| Granularity bit | The granularity bit is zero, which implies that the value of the 16-bit limit is interpreted in units of one byte. |
| B-bit | In a data-segment descriptor, the B-bit is zero, implying that the segment is no larger than 64 Kbytes. |
| D-bit | In an executable-segment descriptor, the D-bit is zero, implying that 16-bit addressing and operands are the default. |

For formats of these descriptors and documentation of their use refer to the iAPX 286 Programmer's Reference Manual.

## 13.2 Two ways to Execute 80286 Tasks

When porting 80286 programs to the 80386, there are two cases to consider:

1. Porting an entire 80286 system to the 80386, complete with 80286 operating system, loader, and system builder.

   In this case, all tasks will have 80286 TSSs. The 80386 is being used as a faster 286.

2. Porting selected 80286 applications to run in an 80386 environment with an 80386 operating system, loader, and system builder.

   In this case, the TSSs used to represent 80286 tasks should be changed to 80386 TSSs. It is theoretically possible to mix 80286 and 80386 TSSs, but the benefits are slight and the problems are great. It is recommended that all tasks in a 80386 software system have 80386 TSSs. It is not necessary to change the 80286 object modules themselves; TSSs are usually constructed by the operating system, by the loader, or by the system builder. Refer to Chapter 16 for further discussion of the interface between 16-bit and 32-bit code.

## 13.3 Differences From 80286

The few differences that do exist primarily affect operating system code.

### 13.3.1 Wraparound of 80286 24-Bit Physical Address Space

With the 80286, any base and offset combination that addresses beyond 16M bytes wraps around to the first megabyte of the 80286 address space. With the 80386, since it has a greater physical address space, any such address falls into the 17th megabyte. In the unlikely event that any software depends on this anomaly, the same effect can be simulated on the 80386 by using paging to map the first 64K bytes of the 17th megabyte of logical addresses to physical addresses in the first megabyte.

### 13.3.2 Reserved Word of Descriptor

Because the 80386 uses the contents of the reserved word (last word) of every descriptor, 80286 programs that place values in this word may not execute correctly on the 80386.

### 13.3.3 New Descriptor Type Codes

Operating-system code that manages space in descriptor tables often uses an invalid value in the access-rights field of descriptor-table entries to identify unused entries. Access rights values of 80H and 00H remain invalid for both the 80286 and 80386. Other values that were invalid on for the 80286 may be valid for the 80386 because of the additional descriptor types defined by the 80386.

### 13.3.4 Restricted Semantics of LOCK

The 80286 processor implements the bus lock function differently than the 80386. Programs that use forms of memory locking specific to the 80286 may not execute properly when transported to a specific application of the 80386.

The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. LOCK may only be used with the following 80386 instructions when they modify memory. An undefined-opcode exception results from using LOCK before any other instruction.

- Bit test and change: BTS, BTR, BTC.
- Exchange: XCHG.
- One-operand arithmetic and logical: INC, DEC, NOT, and NEG.
- Two-operand arithmetic and logical: ADD, ADC, SUB, SBB, AND, OR, XOR.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may lock a larger memory area. For example, typical 8086 and 80286 configurations lock the entire physical memory space. With the 80386, the defined area of memory is guaranteed to be locked against access by a processor executing a locked instruction on exactly the same memory area, i.e., an operand with identical starting address and identical length.

### 13.3.5 Additional Exceptions

The 80386 defines new exceptions that can occur even in systems designed for the 80286.

- Exception #6 —— invalid opcode

  This exception can result from improper use of the LOCK instruction.

- Exception #14 —— page fault

  This exception may occur in an 80286 program if the operating system enables paging. Paging can be used in a system with 80286 tasks as long as all tasks use the same page directory. Because there is no place in an 80286 TSS to store the PDBR, switching to an 80286 task does not change the value of PDBR. Tasks ported from the 80286 should be given 80386 TSSs so they can take full advantage of paging.

# Chapter 14 80386 Real-Address Mode

The real-address mode of the 80386 executes object code designed for execution on 8086, 8088, 80186, or 80188 processors, or for execution in the real-address mode of an 80286:

In effect, the architecture of the 80386 in this mode is almost identical to that of the 8086, 8088, 80186, and 80188. To a programmer, an 80386 in real-address mode appears as a high-speed 8086 with extensions to the instruction set and registers. The principal features of this architecture are defined in Chapters 2 and 3.

This chapter discusses certain additional topics that complete the system programmer's view of the 80386 in real-address mode:

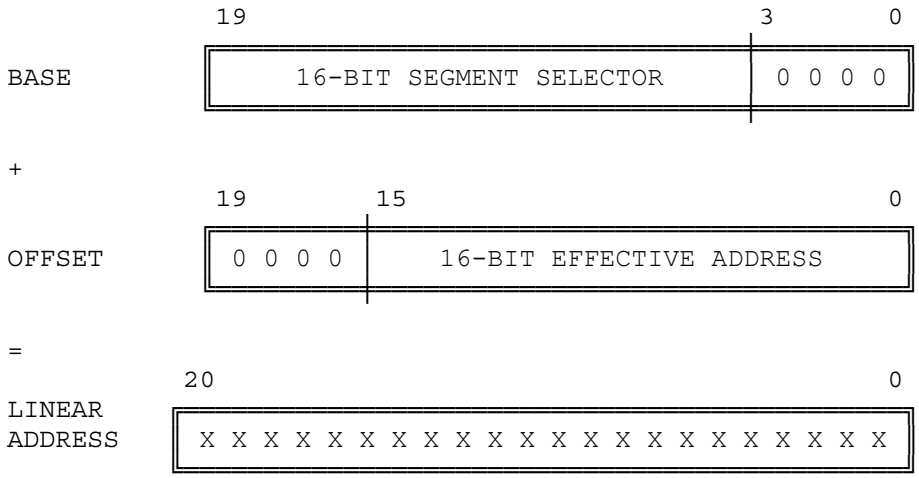
- Address formation.
- Extensions to registers and instructions.
- Interrupt and exception handling.
- Entering and leaving real-address mode.
- Real-address-mode exceptions.
- Differences from 8086.
- Differences from 80286 real-address mode.

## 14.1 Physical Address Formation

The 80386 provides a one Mbyte + 64 Kbyte memory space for an 8086 program. Segment relocation is performed as in the 8086: the 16-bit value in a segment selector is shifted left by four bits to form the base address of a segment. The effective address is extended with four high order zeros and added to the base to form a linear address as Figure 14-1 illustrates. (The linear address is equivalent to the physical address, because paging is not used in real-address mode.) Unlike the 8086, the resulting linear address may have up to 21 significant bits. There is a possibility of a carry when the base address is added to the effective address. On the 8086, the carried bit is truncated, whereas on the 80386 the carried bit is stored in bit position 20 of the linear address.

Unlike the 8086 and 80286, 32-bit effective addresses can be generated (via the address-size prefix); however, the value of a 32-bit address may not exceed 65535 without causing an exception. For full compatibility with 80286 real-address mode, pseudo-protection faults (interrupt 12 or 13 with no error code) occur if an effective address is generated outside the range 0 through 65535.

**Figure 14-1. Real-Address Mode Address Formation**




## 14.2 Registers and Instructions

The register set available in real-address mode includes all the registers defined for the 8086 plus the new registers introduced by the 80386: FS, GS, debug registers, control registers, and test registers. New instructions that explicitly operate on the segment registers FS and GS are available, and the new segment-override prefixes can be used to cause instructions to utilize FS and GS for address calculations. Instructions can utilize 32-bit operands through the use of the operand size prefix.

The instruction codes that cause undefined opcode traps (interrupt 6) include instructions of the protected mode that manipulate or interrogate 80386 selectors and descriptors; namely, VERR, VERW, LAR, LSL, LTR, STR, LLDT, and SLDT. Programs executing in real-address mode are able to take advantage of the new applications-oriented instructions added to the architecture by the introduction of the 80186/80188, 80286 and 80386:

- New instructions introduced by 80186/80188 and 80286.

  — PUSH immediate data
  — Push all and pop all (PUSHA and POPA)
  — Multiply immediate data
  — Shift and rotate by immediate count
  — String I/O
  — ENTER and LEAVE
  — BOUND

- New instructions introduced by 80386.

  — LSS, LFS, LGS instructions
  — Long-displacement conditional jumps
  — Single-bit instructions
  — Bit scan
  — Double-shift instructions
  — Byte set on condition

- Move with sign/zero extension
- Generalized multiply
- MOV to and from control registers
- MOV to and from test registers
- MOV to and from debug registers

## 14.3 Interrupt and Exception Handling

Interrupts and exceptions in 80386 real-address mode work as much as they do on an 8086. Interrupts and exceptions vector to interrupt procedures via an interrupt table. The processor multiplies the interrupt or exception identifier by four to obtain an index into the interrupt table. The entries of the interrupt table are far pointers to the entry points of interrupt or exception handler procedures. When an interrupt occurs, the processor pushes the current values of CS:IP onto the stack, disables interrupts, clears TF (the single-step flag), then transfers control to the location specified in the interrupt table. An IRET instruction at the end of the handler procedure reverses these steps before returning control to the interrupted procedure.

The primary difference in the interrupt handling of the 80386 compared to the 8086 is that the location and size of the interrupt table depend on the contents of the IDTR (IDT register). Ordinarily, this fact is not apparent to programmers, because, after RESET, the IDTR contains a base address of 0 and a limit of 3FFH, which is compatible with the 8086. However, the LIDT instruction can be used in real-address mode to change the base and limit values in the IDTR. Refer to Chapter 9 for details on the IDTR, and the LIDT and SIDT instructions. If an interrupt occurs and the corresponding entry of the interrupt table is beyond the limit stored in the IDTR, the processor raises exception 8.

## 14.4 Entering and Leaving Real-Address Mode

Real-address mode is in effect after a signal on the RESET pin. Even if the system is going to be used in protected mode, the start-up program will execute in real-address mode temporarily while initializing for protected mode.

### 14.4.1 Switching to Protected Mode

The only way to leave real-address mode is to switch to protected mode. The processor enters protected mode when a MOV to CR0 instruction sets the PE (protection enable) bit in CR0. (For compatibility with the 80286, the LMSW instruction may also be used to set the PE bit.)

Refer to Chapter 10 "Initialization" for other aspects of switching to protected mode.

## 14.5 Switching Back to Real-Address Mode

The processor reenters real-address mode if software clears the PE bit in CR0 with a MOV to CR0 instruction. A procedure that attempts to do this, however, should proceed as follows:

1. If paging is enabled, perform the following sequence:

   - Transfer control to linear addresses that have an identity mapping; i.e., linear addresses equal physical addresses.
   - Clear the PG bit in CR0.
   - Move zeros to CR3 to clear out the paging cache.

2. Transfer control to a segment that has a limit of 64K (FFFFH). This loads the CS register with the limit it needs to have in real mode.

3. Load segment registers SS, DS, ES, FS, and GS with a selector that points to a descriptor containing the following values, which are appropriate to real mode:

   - Limit = 64K (FFFFH)
   - Byte granular (G = 0)
   - Expand up (E = 0)
   - Writable (W = 1)
   - Present (P = 1)
   - Base = any value

4. Disable interrupts. A CLI instruction disables INTR interrupts. NMIs can be disabled with external circuitry.

5. Clear the PE bit.

6. Jump to the real mode code to be executed using a far JMP. This action flushes the instruction queue and puts appropriate values in the access rights of the CS register.

7. Use the LIDT instruction to load the base and limit of the real-mode interrupt vector table.

8. Enable interrupts.

9. Load the segment registers as needed by the real-mode code.

## 14.6 Real-Address Mode Exceptions

The 80386 reports some exceptions differently when executing in real-address mode than when executing in protected mode. Table 14-1 details the real-address-mode exceptions.

## 14.7 Differences From 8086

In general, the 80386 in real-address mode will correctly execute ROM-based software designed for the 8086, 8088, 80186, and 80188. Following is a list of the minor differences between 8086 execution on the 80386 and on an 8086.

1. Instruction clock counts.

   The 80386 takes fewer clocks for most instructions than the 8086/8088. The areas most likely to be affected are:

   - Delays required by I/O devices between I/O operations.
   - Assumed delays with 8086/8088 operating in parallel with an 8087.

2. Divide Exceptions Point to the DIV instruction.

   Divide exceptions on the 80386 always leave the saved CS:IP value pointing to the instruction that failed. On the 8086/8088, the CS:IP value points to the next instruction.

3. Undefined 8086/8088 opcodes.

   Opcodes that were not defined for the 8086/8088 will cause exception 6 or will execute one of the new instructions defined for the 80386.

4. Value written by PUSH SP.

   The 80386 pushes a different value on the stack for PUSH SP than the 8086/8088. The 80386 pushes the value of SP before SP is incremented as part of the push operation; the 8086/8088 pushes the value of SP after it is incremented. If the value pushed is important, replace PUSH SP instructions with the following three instructions:

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

   This code functions as the 8086/8088 PUSH SP instruction on the 80386.

5. Shift or rotate by more than 31 bits.

   The 80386 masks all shift and rotate counts to the low-order five bits. This MOD 32 operation limits the count to a maximum of 31 bits, thereby limiting the time that interrupt response is delayed while the instruction is executing.

6. Redundant prefixes.

   The 80386 sets a limit of 15 bytes on instruction length. The only way to violate this limit is by putting redundant prefixes before an instruction. Exception 13 occurs if the limit on instruction length is violated. The 8086/8088 has no instruction length limit.

7. Operand crossing offset 0 or 65,535.

   On the 8086, an attempt to access a memory operand that crosses offset 65,535 (e.g., MOV a word to offset 65,535) or offset 0 (e.g., PUSH a word when SP = 1) causes the offset to wrap around modulo 65,536. The 80386 raises an exception in these cases——exception 13 if the segment is a data segment (i.e., if CS, DS, ES, FS, or GS is being used to address the segment), exception 12 if the segment is a stack segment (i.e., if SS is being used).

8. Sequential execution across offset 65,535.

   On the 8086, if sequential execution of instructions proceeds past offset 65,535, the processor fetches the next instruction byte from offset 0 of the same segment. On the 80386, the processor raises exception 13 in such a case.

9. LOCK is restricted to certain instructions.

   The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. The 80386 always asserts the LOCK signal during an XCHG instruction with memory (even if the LOCK prefix is not used). LOCK may only be used with the following 80386 instructions when they update memory: BTS, BTR, BTC, XCHG, ADD, ADC, SUB, SBB, INC, DEC, AND, OR, XOR, NOT, and NEG. An undefined-opcode exception (interrupt 6) results from using LOCK before any other instruction.

10. Single-stepping external interrupt handlers.

    The priority of the 80386 single-step exception is different from that of the 8086/8088. The change prevents an external interrupt handler from being single-stepped if the interrupt occurs while a program is being single-stepped. The 80386 single-step exception has higher priority that any external interrupt. The 80386 will still single-step through an interrupt handler invoked by the INT instructions or by an exception.

11. IDIV exceptions for quotients of 80H or 8000H.

    The 80386 can generate the largest negative number as a quotient for the IDIV instruction. The 8086/8088 causes exception zero instead.

12. Flags in stack.

    The setting of the flags stored by PUSHF, by interrupts, and by exceptions is different from that stored by the 8086 in bit positions 12 through 15. On the 8086 these bits are stored as ones, but in 80386 real-address mode bit 15 is always zero, and bits 14 through 12 reflect the last value loaded into them.

13. NMI interrupting NMI handlers.

    After an NMI is recognized on the 80386, the NMI interrupt is masked until an IRET instruction is executed.

14. Coprocessor errors vector to interrupt 16.

    Any 80386 system with a coprocessor must use interrupt vector 16 for the coprocessor error exception. If an 8086/8088 system uses another vector for the 8087 interrupt, both vectors should point to the coprocessor-error exception handler.

15. Numeric exception handlers should allow prefixes.

    On the 80386, the value of CS:IP saved for coprocessor exceptions points at any prefixes before an ESC instruction. On 8086/8088 systems, the saved CS:IP points to the ESC instruction.

16. Coprocessor does not use interrupt controller.

    The coprocessor error signal to the 80386 does not pass through an interrupt controller (an 8087 INT signal does). Some instructions in a coprocessor error handler may need to be deleted if they deal with the interrupt controller.

17. Six new interrupt vectors.

    The 80386 adds six exceptions that arise only if the 8086 program has a hidden bug. It is recommended that exception handlers be added that treat these exceptions as invalid operations. This additional software does not significantly affect the existing 8086 software because the interrupts do not normally occur. These interrupt identifiers should not already have been used by the 8086 software, because they are in the range reserved by Intel. Table 14-2 describes the new 80386 exceptions.

18. One megabyte wraparound.

    The 80386 does not wrap addresses at 1 megabyte in real-address mode. On members of the 8086 family, it possible to specify addresses greater than one megabyte. For example, with a selector value 0FFFFH and an offset of 0FFFFH, the effective address would be 10FFEFH (1 Mbyte + 65519). The 8086, which can form adresses only up to 20 bits long, truncates the high-order bit, thereby "wrapping" this address to 0FFEFH. However, the 80386, which can form addresses up to 32 bits long does not truncate such an address.

**Table 14-1. 80386 Real-Address Mode Exceptions**

| Description | Interrupt Number | Function that Can Generate the Exception | Return Address Points to Faulting Instruction |
|---|---|---|---|
| Divide error | 0 | DIV, IDIV | YES |
| Debug exceptions | 1 | All | Some debug exceptions point to the faulting instruction, others to the next instruction. The exception handler can determine which has occurred by examining DR6. |
| Breakpoint | 3 | INT | NO |
| Overflow | 4 | INTO | NO |
| Bounds check | 5 | BOUND | YES |
| Invalid opcode | 6 | Any undefined opcode or LOCK used with wrong instruction | YES |
| Coprocessor not available | 7 | ESC or WAIT | YES |
| Interrupt table limit too small | 8 | INT vector is not within IDTR limit | YES |
| Reserved | 9-12 | | |
| Stack fault | 12 | Memory operand crosses offset 0 or 0FFFFH | YES |
| Pseudo-protection exception | 13 | Memory operand crosses offset 0FFFFH or attempt to execute past offset 0FFFFH or instruction longer than 15 bytes | YES |
| Reserved | 14,15 | | |
| Coprocessor error | 16 | ESC or WAIT | YES |
| Coprocessor errors are reported on the first ESC or WAIT instruction after the ESC instruction that caused the error. | | | |
| Two-byte SW interrupt | 0-255 | INT n | NO |

**Table 14-2. New 80386 Exceptions**

| Interrupt Identifier | Function |
|---|---|
| 5 | A BOUND instruction was executed with a register value outside the limit values. |
| 6 | An undefined opcode was encountered or LOCK was used improperly before an instruction to which it does not apply. |
| 7 | The EM bit in the MSW is set when an ESC instruction was encountered. This exception also occurs on a WAIT instruction if TS is set. |
| 8 | An exception or interrupt has vectored to an interrupt table entry beyond the interrupt table limit in IDTR. This can occur only if the LIDT instruction has changed the limit from the default value of 3FFH, which is enough for all 256 interrupt IDs. |
| 12 | Operand crosses extremes of stack segment, e.g., MOV operation at offset 0FFFFH or push with SP=1 during PUSH, CALL, or INT. |
| 13 | Operand crosses extremes of a segment other than a stack segment; or sequential instruction execution attempts to proceed beyond offset 0FFFFH; or an instruction is longer than 15 bytes (including prefixes). |

## 14.8 Differences From 80286 Real-Address Mode

The few differences that exist between 80386 real-address mode and 80286 real-address mode are not likely to affect any existing 80286 programs except possibly the system initialization procedures.

### 14.8.1 Bus Lock

The 80286 processor implements the bus lock function differently than the 80386. Programs that use forms of memory locking specific to the 80286 may not execute properly if transported to a specific application of the 80386.

The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. LOCK may only be used with the following 80386 instructions when they modify memory. An undefined-opcode exception results from using LOCK before any other instruction.

- Bit test and change: BTS, BTR, BTC.
- Exchange: XCHG.
- One-operand arithmetic and logical: INC, DEC, NOT, and NEG.
- Two-operand arithmetic and logical: ADD, ADC, SUB, SBB, AND, OR, XOR.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may lock a larger memory area. For example, typical 8086 and 80286 configurations lock the entire physical memory space. With the 80386, the defined area of memory is guranteed to be locked against access by a processor executing a locked instruction on exactly the same memory area, i.e., an operand with identical starting address and identical length.

### 14.8.2 Location of First Instruction

The starting location is 0FFFFFFF0H (sixteen bytes from end of 32-bit address space) on the 80386 rather than 0FFFFF0H (sixteen bytes from end of 24-bit address space) as on the 80286. Many 80286 ROM initialization programs will work correctly in this new environment. Others can be made to work correctly with external hardware that redefines the signals on A{31-20}.

### 14.8.3 Initial Values of General Registers

On the 80386, certain general registers may contain different values after RESET than on the 80286. This should not cause compatibility problems, because the content of 8086 registers after RESET is undefined. If self-test is requested during the reset sequence and errors are detected in the 80386 unit, EAX will contain a nonzero value. EDX contains the component and revision identifier. Refer to Chapter 10 for more information.

### 14.8.4 MSW Initialization

The 80286 initializes the MSW register to FFF0H, but the 80386 initializes this register to 0000H. This difference should have no effect, because the bits that are different are undefined on the 80286. Programs that read the value of the MSW will behave differently on the 80386 only if they depend on the setting of the undefined, high-order bits.

# Chapter 15 Virtual 8086 Mode

The 80386 supports execution of one or more 8086, 8088, 80186, or 80188 programs in an 80386 protected-mode environment. An 8086 program runs in this environment as part of a V86 (virtual 8086) task. V86 tasks take advantage of the hardware support of multitasking offered by the protected mode. Not only can there be multiple V86 tasks, each one executing an 8086 program, but V86 tasks can be multiprogrammed with other 80386 tasks.

The purpose of a V86 task is to form a "virtual machine" with which to execute an 8086 program. A complete virtual machine consists not only of 80386 hardware but also of systems software. Thus, the emulation of an 8086 is the result of cooperation between hardware and software:

- The hardware provides a virtual set of registers (via the TSS), a virtual memory space (the first megabyte of the linear address space of the task), and directly executes all instructions that deal with these registers and with this address space.

- The software controls the external interfaces of the virtual machine (I/O, interrupts, and exceptions) in a manner consistent with the larger environment in which it executes. In the case of I/O, software can choose either to emulate I/O instructions or to let the hardware execute them directly without software intervention.

Software that helps implement virtual 8086 machines is called a V86 monitor.

## 15.1 Executing 8086 Code

The processor executes in V86 mode when the VM (virtual machine) bit in the EFLAGS register is set. The processor tests this flag under two general conditions:

1. When loading segment registers to know whether to use 8086-style address formation.

2. When decoding instructions to determine which instructions are sensitive to IOPL.

Except for these two modifications to its normal operations, the 80386 in V86 mode operated much as in protected mode.

### 15.1.1 Registers and Instructions

The register set available in V86 mode includes all the registers defined for the 8086 plus the new registers introduced by the 80386: FS, GS, debug registers, control registers, and test registers. New instructions that explicitly operate on the segment registers FS and GS are available, and the new segment-override prefixes can be used to cause instructions to utilize FS and GS for address calculations. Instructions can utilize 32-bit operands through the use of the operand size prefix.

8086 programs running as V86 tasks are able to take advantage of the new applications-oriented instructions added to the architecture by the introduction of the 80186/80188, 80286 and 80386:

- New instructions introduced by 80186/80188 and 80286.
  - PUSH immediate data
  - Push all and pop all (PUSHA and POPA)
  - Multiply immediate data
  - Shift and rotate by immediate count
  - String I/O
  - ENTER and LEAVE
  - BOUND

- New instructions introduced by 80386.
  - LSS, LFS, LGS instructions
  - Long-displacement conditional jumps
  - Single-bit instructions
  - Bit scan
  - Double-shift instructions
  - Byte set on condition
  - Move with sign/zero extension
  - Generalized multiply

### 15.1.2 Linear Address Formation

In V86 mode, the 80386 processor does not interpret 8086 selectors by referring to descriptors; instead, it forms linear addresses as an 8086 would. It shifts the selector left by four bits to form a 20-bit base address. The effective address is extended with four high-order zeros and added to the base address to create a linear address as Figure 15-1 illustrates.

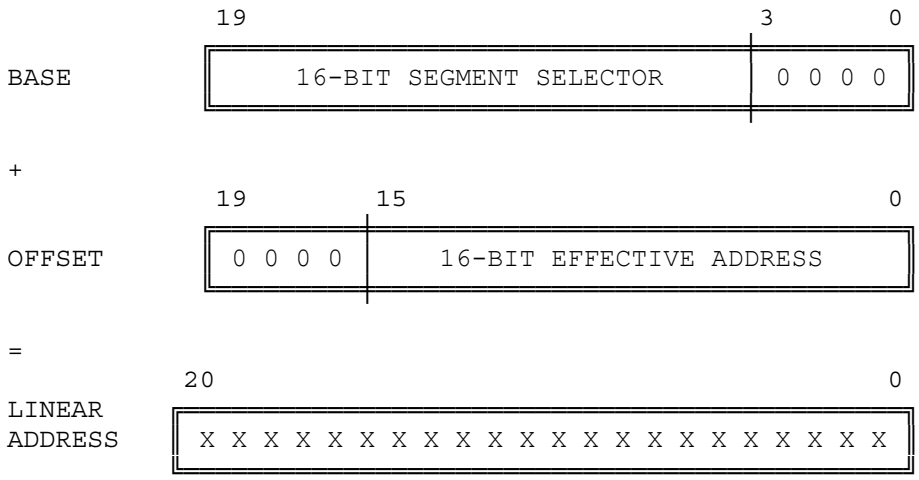
Because of the possibility of a carry, the resulting linear address may contain up to 21 significant bits. An 8086 program may generate linear addresses anywhere in the range 0 to 10FFEFH (one megabyte plus approximately 64 Kbytes) of the task's linear address space.

V86 tasks generate 32-bit linear addresses. While an 8086 program can only utilize the low-order 21 bits of a linear address, the linear address can be mapped via page tables to any 32-bit physical address.

Unlike the 8086 and 80286, 32-bit effective addresses can be generated (via the address-size prefix); however, the value of a 32-bit address may not

exceed 65,535 without causing an exception. For full compatibility with 80286 real-address mode, pseudo-protection faults (interrupt 12 or 13 with no error code) occur if an address is generated outside the range 0 through 65,535.

**Figure 15-1. V86 Mode Address Formation**




## 15.2 Structure of a V86 Task

A V86 task consists partly of the 8086 program to be executed and partly of 80386 "native mode" code that serves as the virtual-machine monitor. The task must be represented by an 80386 TSS (not an 80286 TSS). The processor enters V86 mode to execute the 8086 program and returns to protected mode to execute the monitor or other 80386 tasks.

To run successfully in V86 mode, an existing 8086 program needs the following:

- A V86 monitor.
- Operating-system services.

The V86 monitor is 80386 protected-mode code that executes at privilege-level zero. The monitor consists primarily of initialization and exception-handling procedures. As for any other 80386 program, executable-segment descriptors for the monitor must exist in the GDT or in the task's LDT. The linear addresses above 10FFEFH are available for the V86 monitor, the operating system, and other systems software. The monitor may also need data-segment descriptors so that it can examine the interrupt vector table or other parts of the 8086 program in the first megabyte of the address space.

In general, there are two options for implementing the 8086 operating system:

1. The 8086 operating system may run as part of the 8086 code. This approach is desirable for any of the following reasons:

   - The 8086 applications code modifies the operating system.
   - There is not sufficient development time to reimplement the 8086 operating system as 80386 code.

2. The 8086 operating system may be implemented or emulated in the V86 monitor. This approach is desirable for any of the following reasons:

   - Operating system functions can be more easily coordinated among several V86 tasks.
   - The functions of the 8086 operating system can be easily emulated by calls to the 80386 operating system.

Note that, regardless of the approach chosen for implementing the 8086 operating system, different V86 tasks may use different 8086 operating systems.

### 15.2.1 Using Paging for V86 Tasks

Paging is not necessary for a single V86 task, but paging is useful or necessary for any of the following reasons:

- To create multiple V86 tasks. Each task must map the lower megabyte of linear addresses to different physical locations.
- To emulate the megabyte wrap. On members of the 8086 family, it is possible to specify addresses larger than one megabyte. For example, with a selector value of 0FFFFH and an offset of 0FFFFH, the effective address would be 10FFEFH (one megabyte + 65519). The 8086, which can form addresses only up to 20 bits long, truncates the high-order bit, thereby "wrapping" this address to 0FFEFH. The 80386, however, which can form addresses up to 32 bits long does not truncate such an address. If any 8086 programs depend on this addressing anomaly, the same effect can be achieved in a V86 task by mapping linear addresses between 100000H and 110000H and linear addresses between 0 and 10000H to the same physical addresses.
- To create a virtual address space larger than the physical address space.
- To share 8086 OS code or ROM code that is common to several 8086 programs that are executing simultaneously.
- To redirect or trap references to memory-mapped I/O devices.

### 15.2.2 Protection within a V86 Task

Because it does not refer to descriptors while executing 8086 programs, the processor also does not utilize the protection mechanisms offered by descriptors. To protect the systems software that runs in a V86 task from the 8086 program, software designers may follow either of these approaches:

- Reserve the first megabyte (plus 64 kilobytes) of each task's linear address space for the 8086 program. An 8086 task cannot generate addresses outside this range.

- Use the U/S bit of page-table entries to protect the virtual-machine monitor and other systems software in each virtual 8086 task's space. When the processor is in V86 mode, CPL is 3. Therefore, an 8086 program has only user privileges. If the pages of the virtual-machine monitor have supervisor privilege, they cannot be accessed by the 8086 program.

## 15.3 Entering and Leaving V86 Mode

Figure 15-2 summarizes the ways that the processor can enter and leave an 8086 program. The processor can enter V86 by either of two means:

1. A task switch to an 80386 task loads the image of EFLAGS from the new TSS. The TSS of the new task must be an 80386 TSS, not an 80286 TSS, because the 80286 TSS does not store the high-order word of EFLAGS, which contains the VM flag. A value of one in the VM bit of the new EFLAGS indicates that the new task is executing 8086 instructions; therefore, while loading the segment registers from the TSS, the processor forms base addresses as the 8086 would.

2. An IRET from a procedure of an 80386 task loads the image of EFLAGS from the stack. A value of one in VM in this case indicates that the procedure to which control is being returned is an 8086 procedure. The CPL at the time the IRET is executed must be zero, else the processor does not change VM.

The processor leaves V86 mode when an interrupt or exception occurs. There are two cases:

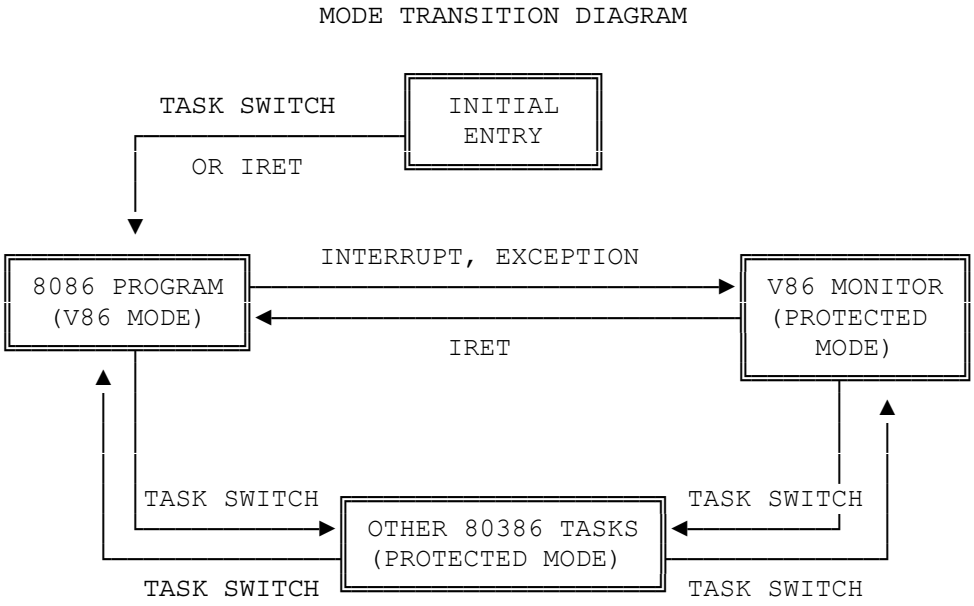
1. The interrupt or exception causes a task switch. A task switch from a V86 task to any other task loads EFLAGS from the TSS of the new task. If the new TSS is an 80386 TSS and the VM bit in the EFLAGS image is zero or if the new TSS is an 80286 TSS, then the processor clears the VM bit of EFLAGS, loads the segment registers from the new TSS using 80386-style address formation, and begins executing the instructions of the new task according to 80386 protected-mode semantics.

2. The interrupt or exception vectors to a privilege-level zero procedure. The processor stores the current setting of EFLAGS on the stack, then clears the VM bit. The interrupt or exception handler, therefore, executes as "native" 80386 protected-mode code. If an interrupt or exception vectors to a conforming segment or to a privilege level other than three, the processor causes a

general-protection exception; the error code is the selector of the executable segment to which transfer was attempted.

Systems software does not manipulate the VM flag directly, but rather manipulates the image of the EFLAGS register that is stored on the stack or in the TSS. The V86 monitor sets the VM flag in the EFLAGS image on the stack or in the TSS when first creating a V86 task. Exception and interrupt handlers can examine the VM flag on the stack. If the interrupted procedure was executing in V86 mode, the handler may need to invoke the V86 monitor.

**Figure 15-2. Entering and Leaving the 8086 Program**




### 15.3.1 Transitions Through Task Switches

A task switch to or from a V86 task may be due to any of three causes:

1. An interrupt that vectors to a task gate.
2. An action of the scheduler of the 80386 operating system.
3. An IRET when the NT flag is set.

In any of these cases, the processor changes the VM bit in EFLAGS according to the image of EFLAGS in the new TSS. If the new TSS is an 80286 TSS, the high-order word of EFLAGS is not in the TSS; the processor clears VM in this case. The processor updates VM prior to loading the segment registers from the images in the new TSS. The new setting of VM determines whether the processor interprets the new segment-register images as 8086 selectors or 80386/80286 selectors.

### 15.3.2 Transitions Through Trap Gates and Interrupt Gates

The processor leaves V86 mode as the result of an exception or interrupt that vectors via a trap or interrupt gate to a privilege-level zero procedure. The exception or interrupt handler returns to the 8086 code by executing an IRET.

Because it was designed for execution by an 8086 processor, an 8086 program in a V86 task will have an 8086-style interrupt table starting at linear address zero. However, the 80386 does not use this table directly. For all exceptions and interrupts that occur in V86 mode, the processor vectors through the IDT. The IDT entry for an interrupt or exception that occurs in a V86 task must contain either:

- A task gate.
- An 80386 trap gate (type 14) or an 80386 interrupt gate (type 15), which must point to a nonconforming, privilege-level zero, code segment.

Interrupts and exceptions that have 80386 trap or interrupt gates in the IDT vector to the appropriate handler procedure at privilege-level zero. The contents of all the 8086 segment registers are stored on the PL 0 stack. Figure 15-3 shows the format of the PL 0 stack after an exception or interrupt that occurs while a V86 task is executing an 8086 program.

After the processor stores all the 8086 segment registers on the PL 0 stack, it loads all the segment registers with zeros before starting to execute the handler procedure. This permits the interrupt handler to safely save and restore the DS, ES, FS, and GS registers as 80386 selectors. Interrupt handlers that may be invoked in the context of either a regular task or a V86 task, can use the same prolog and epilog code for register saving regardless of the kind of task. Restoring zeros to these registers before execution of the IRET does not cause a trap in the interrupt handler. Interrupt procedures that expect values in the segment registers or that return values via segment registers have to use the register images stored on the PL 0 stack. Interrupt handlers that need to know whether the interrupt occurred in V86 mode can examine the VM bit in the stored EFLAGS image.

An interrupt handler passes control to the V86 monitor if the VM bit is set in the EFLAGS image stored on the stack and the interrupt or exception is one that the monitor needs to handle. The V86 monitor may either:

- Handle the interrupt completely within the V86 monitor.
- Invoke the 8086 program's interrupt handler.

Reflecting an interrupt or exception back to the 8086 code involves the following steps:

1. Refer to the 8086 interrupt vector to locate the appropriate handler procedure.
2. Store the state of the 8086 program on the privilege-level three stack.

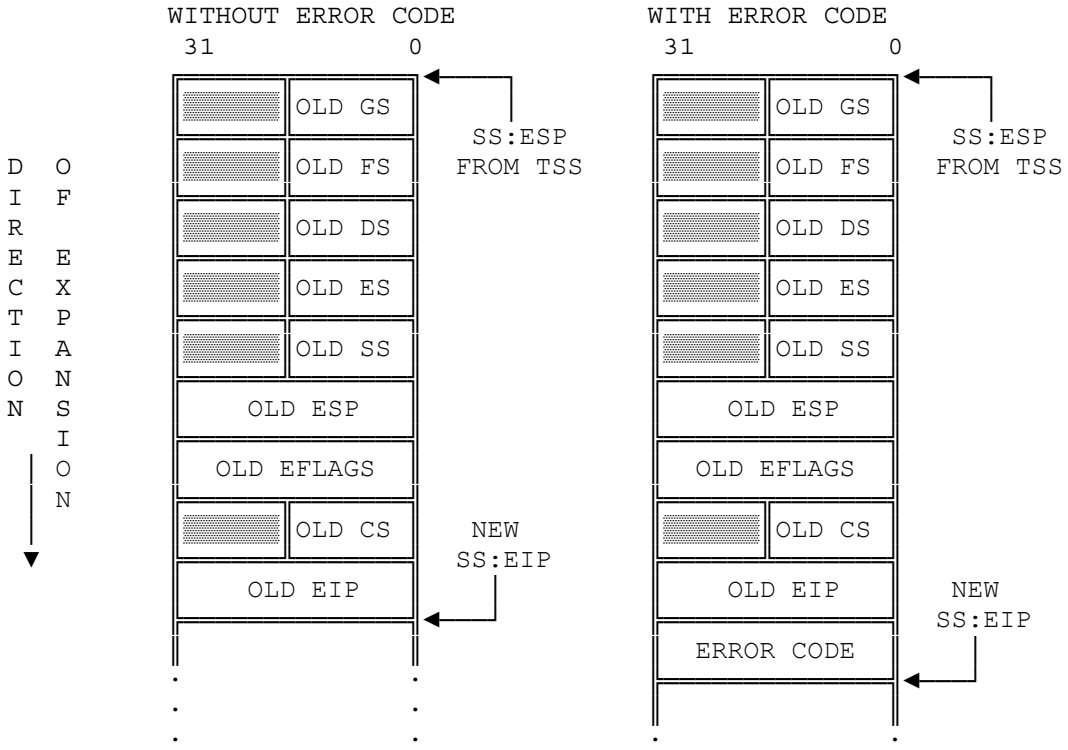
3. Change the return link on the privilege-level zero stack to point to the privilege-level three handler procedure.

4. Execute an IRET so as to pass control to the handler.

5. When the IRET by the privilege-level three handler again traps to the V86 monitor, restore the return link on the privilege-level zero stack to point to the originally interrupted, privilege-level three procedure.

6. Execute an IRET so as to pass control back to the interrupted procedure.

**Figure 15-3. PL 0 Stack after Interrupt in V86 Task**




## 15.4 Additional Sensitive Instructions

When the 80386 is executing in V86 mode, the instructions PUSHF, POPF, INT n, and IRET are sensitive to IOPL. The instructions IN, INS, OUT, and OUTS, which are ordinarily sensitive in protected mode, are not sensitive in V86 mode. Following is a complete list of instructions that are sensitive in V86 mode:

```
CLI     —— Clear Interrupt-Enable Flag
STI     —— Set Interrupt-Enable Flag
LOCK    —— Assert Bus-Lock Signal
PUSHF   —— Push Flags
POPF    —— Pop Flags
INT n   —— Software Interrupt
RET     —— Interrupt Return
```

CPL is always three in V86 mode; therefore, if IOPL < 3, these instructions will trigger a general-protection exceptions. These instructions are made sensitive so that their functions can be simulated by the V86 monitor.

### 15.4.1 Emulating 8086 Operating System Calls

INT n is sensitive so that the V86 monitor can intercept calls to the 8086 OS. Many 8086 operating systems are called by pushing parameters onto the stack, then executing an INT n instruction. If IOPL < 3, INT n instructions will be intercepted by the V86 monitor. The V86 monitor can then emulate the function of the 8086 operating system or reflect the interrupt back to the 8086 operating system in V86 mode.

### 15.4.2 Virtualizing the Interrupt-Enable Flag

When the processor is executing 8086 code in a V86 task, the instructions PUSHF, POPF, and IRET are sensitive to IOPL so that the V86 monitor can control changes to the interrupt-enable flag (IF). Other instructions that affect IF (STI and CLI) are IOPL sensitive both in 8086 code and in 80386/80386 code.

Many 8086 programs that were designed to execute on single-task systems set and clear IF to control interrupts. However, when these same programs are executed in a multitasking environment, such control of IF can be disruptive. If IOPL is less than three, all instructions that change or interrogate IF will trap to the V86 monitor. The V86 monitor can then control IF in a manner that both suits the needs of the larger environment and is transparent to the 8086 program.

## 15.5 Virtual I/O

Many 8086 programs that were designed to execute on single-task systems use I/O devices directly. However, when these same programs are executed in a multitasking environment, such use of devices can be disruptive. The 80386 provides sufficient flexibility to control I/O in a manner that both suits the needs of the new environment and is transparent to the 8086 program. Designers may take any of several possible approaches to controlling I/O:

- Implement or emulate the 8086 operating system as an 80386 program and require the 8086 application to do I/O via software interrupts to the operating system, trapping all attempts to do I/O directly.

- Let the 8086 program take complete control of all I/O.
- Selectively trap and emulate references that a task makes to specific I/O ports.
- Trap or redirect references to memory-mapped I/O addresses.

The method of controlling I/O depends upon whether I/O ports are I/O mapped or memory mapped.

### 15.5.1 I/O-Mapped I/O

I/O-mapped I/O in V86 mode differs from protected mode only in that the protection mechanism does not consult IOPL when executing the I/O instructions IN, INS, OUT, OUTS. Only the I/O permission bit map controls the right for V86 tasks to execute these I/O instructions.

The I/O permission map traps I/O instructions selectively depending on the I/O addresses to which they refer. The I/O permission bit map of each V86 task determines which I/O addresses are trapped for that task. Because each task may have a different I/O permission bit map, the addresses trapped for one task may be different from those trapped for others. Refer to Chapter 8 for more information about the I/O permission map.

### 15.5.2 Memory-Mapped I/O

In hardware designs that utilize memory-mapped I/O, the paging facilities of the 80386 can be used to trap or redirect I/O operations. Each task that executes memory-mapped I/O must have a page (or pages) for the memory-mapped address space. The V86 monitor may control memory-mapped I/O by any of these means:

- Assign the memory-mapped page to appropriate physical addresses. Different tasks may have different physical addresses, thereby preventing the tasks from interfering with each other.
- Cause a trap to the monitor by forcing a page fault on the memory-mapped page. Read-only pages trap writes. Not-present pages trap both reads and writes.

Intervention for every I/O might be excessive for some kinds of I/O devices. A page fault can still be used in this case to cause intervention on the first I/O operation. The monitor can then at least make sure that the task has exclusive access to the device. Then the monitor can change the page status to present and read/write, allowing subsequent I/O to proceed at full speed.

### 15.5.3 Special I/O Buffers

Buffers of intelligent controllers (for example, a bit-mapped graphics buffer) can also be virtualized via page mapping. The linear space for the buffer can be mapped to a different physical space for each virtual 8086 task. The V86 monitor can then assume responsibility for spooling the data or assigning the virtual buffer to the real buffer at appropriate times.

## 15.6 Differences From 8086

In general, V86 mode will correctly execute software designed for the 8086, 8088, 80186, and 80188. Following is a list of the minor differences between 8086 execution on the 80386 and on an 8086.

1. Instruction clock counts.

   The 80386 takes fewer clocks for most instructions than the 8086/8088. The areas most likely to be affected are:

   - Delays required by I/O devices between I/O operations.
   - Assumed delays with 8086/8088 operating in parallel with an 8087.

2. Divide exceptions point to the DIV instruction.

   Divide exceptions on the 80386 always leave the saved CS:IP value pointing to the instruction that failed. On the 8086/8088, the CS:IP value points to the next instruction.

3. Undefined 8086/8088 opcodes.

   Opcodes that were not defined for the 8086/8088 will cause exception 6 or will execute one of the new instructions defined for the 80386.

4. Value written by PUSH SP.

   The 80386 pushes a different value on the stack for PUSH SP than the 8086/8088. The 80386 pushes the value of SP before SP is incremented as part of the push operation; the 8086/8088 pushes the value of SP after it is incremented. If the value pushed is important, replace PUSH SP instructions with the following three instructions:

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

   This code functions as the 8086/8088 PUSH SP instruction on the 80386.

5. Shift or rotate by more than 31 bits.

   The 80386 masks all shift and rotate counts to the low-order five bits. This MOD 32 operation limits the count to a maximum of 31 bits, thereby limiting the time that interrupt response is delayed while the instruction is executing.

6. Redundant prefixes.

   The 80386 sets a limit of 15 bytes on instruction length. The only way to violate this limit is by putting redundant prefixes before an instruction. Exception 13 occurs if the limit on instruction length is violated. The 8086/8088 has no instruction length limit.

7. Operand crossing offset 0 or 65,535.

   On the 8086, an attempt to access a memory operand that crosses offset 65,535 (e.g., MOV a word to offset 65,535) or offset 0 (e.g., PUSH a word when SP = 1) causes the offset to wrap around modulo 65,536. The 80386 raises an exception in these cases——exception 13 if the segment is a data segment (i.e., if CS, DS, ES, FS, or GS is being used to address the segment), exception 12 if the segment is a stack segment (i.e., if SS is being used).

8. Sequential execution across offset 65,535.

   On the 8086, if sequential execution of instructions proceeds past offset 65,535, the processor fetches the next instruction byte from offset 0 of the same segment. On the 80386, the processor raises exception 13 in such a case.

9. LOCK is restricted to certain instructions.

   The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. The 80386 always asserts the LOCK signal during an XCHG instruction with memory (even if the LOCK prefix is not used). LOCK may only be used with the following 80386 instructions when they update memory: BTS, BTR, BTC, XCHG, ADD, ADC, SUB, SBB, INC, DEC, AND, OR, XOR, NOT, and NEG. An undefined-opcode exception (interrupt 6) results from using LOCK before any other instruction.

10. Single-stepping external interrupt handlers.

    The priority of the 80386 single-step exception is different from that of the 8086/8088. The change prevents an external interrupt handler from being single-stepped if the interrupt occurs while a program is being single-stepped. The 80386 single-step exception has higher priority that any external interrupt. The 80386 will still single-step through an interrupt handler invoked by the INT instructions or by an exception.

11. IDIV exceptions for quotients of 80H or 8000H.

    The 80386 can generate the largest negative number as a quotient for the IDIV instruction. The 8086/8088 causes exception zero instead.

12. Flags in stack.

    The setting of the flags stored by PUSHF, by interrupts, and by exceptions is different from that stored by the 8086 in bit positions 12 through 15. On the 8086 these bits are stored as ones, but in V86 mode bit 15 is always zero, and bits 14 through 12 reflect the last value loaded into them.

13. NMI interrupting NMI handlers.

    After an NMI is recognized on the 80386, the NMI interrupt is masked until an IRET instruction is executed.

14. Coprocessor errors vector to interrupt 16.

    Any 80386 system with a coprocessor must use interrupt vector 16 for the coprocessor error exception. If an 8086/8088 system uses another vector for the 8087 interrupt, both vectors should point to the coprocessor-error exception handler.

15. Numeric exception handlers should allow prefixes.

    On the 80386, the value of CS:IP saved for coprocessor exceptions points at any prefixes before an ESC instruction. On 8086/8088 systems, the saved CS:IP points to the ESC instruction itself.

16. Coprocessor does not use interrupt controller.

    The coprocessor error signal to the 80386 does not pass through an interrupt controller (an 8087 INT signal does). Some instructions in a coprocessor error handler may need to be deleted if they deal with the interrupt controller.

## 15.7 Differences From 80286 Real-Address Mode

The 80286 processor implements the bus lock function differently than the 80386. This fact may or may not be apparent to 8086 programs, depending on how the V86 monitor handles the LOCK prefix. LOCKed instructions are sensitive to IOPL; therefore, software designers can choose to emulate its function. If, however, 8086 programs are allowed to execute LOCK directly, programs that use forms of memory locking specific to the 8086 may not execute properly when transported to a specific application of the 80386.

The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. LOCK may only be used with the following 80386 instructions when they modify memory. An undefined-opcode exception results from using LOCK before any other instruction.

- Bit test and change: BTS, BTR, BTC.
- Exchange: XCHG.
- One-operand arithmetic and logical: INC, DEC, NOT, and NEG.
- Two-operand arithmetic and logical: ADD, ADC, SUB, SBB, AND, OR, XOR.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may lock a larger memory area. For example, typical 8086 and 80286 configurations lock the entire physical memory space. With the 80386, the defined area of memory is guaranteed to be locked against access by a processor executing a locked instruction on exactly the same memory area, i.e., an operand with identical starting address and identical length.

# Chapter 16 Mixing 16-Bit and 32 Bit Code

The 80386 running in protected mode is a 32-bit microprocessor, but it is designed to support 16-bit processing at three levels:

1. Executing 8086/80286 16-bit programs efficiently with complete compatibility.

2. Mixing 16-bit modules with 32-bit modules.

3. Mixing 16-bit and 32-bit addresses and operands within one module.

The first level of support for 16-bit programs has already been discussed in Chapter 13, Chapter 14, and Chapter 15. This chapter shows how 16-bit and 32-bit modules can cooperate with one another, and how one module can utilize both 16-bit and 32-bit operands and addressing.

The 80386 functions most efficiently when it is possible to distinguish between pure 16-bit modules and pure 32-bit modules. A pure 16-bit module has these characteristics:

- All segments occupy 64 Kilobytes or less.
- Data items are either 8 bits or 16 bits wide.
- Pointers to code and data have 16-bit offsets.
- Control is transferred only among 16-bit segments.

A pure 32-bit module has these characteristics:

- Segments may occupy more than 64 Kilobytes (zero bytes to 4 gigabytes).

- Data items are either 8 bits or 32 bits wide.

- Pointers to code and data have 32-bit offsets.

- Control is transferred only among 32-bit segments.

Pure 16-bit modules do exist; they are the modules designed for 16-bit microprocessors. Pure 32-bit modules may exist in new programs designed explicitly for the 80386. However, as systems designers move applications from 16-bit processors to the 32-bit 80386, it will not always be possible to maintain these ideals of pure 16-bit or 32-bit modules. It may be expedient to execute old 16-bit modules in a new 32-bit environment without making source-code changes to the old modules if any of the following conditions is true:

- Modules will be converted one-by-one from 16-bit environments to 32-bit environments.

- Older, 16-bit compilers and software-development tools will be utilized in the new32-bit operating environment until new 32-bit versions can be created.

- The source code of 16-bit modules is not available for modification.
- The specific data structures used by a given module inherently utilize 16-bit words.
- The native word size of the source language is 16 bits.

On the 80386, 16-bit modules can be mixed with 32-bit modules. To design a system that mixes 16- and 32-bit code requires an understanding of the mechanisms that the 80386 uses to invoke and control its 32-bit and 16-bit features.

## 16.1 How the 80386 Implements 16-Bit and 32-Bit Features

The features of the architecture that permit the 80386 to work equally well with 32-bit and 16-bit address and operand sizes include:

- The D-bit (default bit) of code-segment descriptors, which determines the default choice of operand-size and address-size for the instructions of a code segment. (In real-address mode and V86 mode, which do not use descriptors, the default is 16 bits.) A code segment whose D-bit is set is known as a USE32 segment; a code segment whose D-bit is zero is a USE16 segment. The D-bit eliminates the need to encode the operand size and address size in instructions when all instructions use operands and effective addresses of the same size.
- Instruction prefixes that explicitly override the default choice of operand size and address size (available in protected mode as well as in real-address mode and V86 mode).
- Separate 32-bit and 16-bit gates for intersegment control transfers (including call gates, interrupt gates, and trap gates). The operand size for the control transfer is determined by the type of gate, not by the D-bit or prefix of the transfer instruction.
- Registers that can be used both for 32-bit and 16-bit operands and effective-address calculations.
- The B-bit (big bit) of data-segment descriptors, which determines the size of stack pointer (32-bit ESP or 16-bit SP) used by the CPU for implicit stack references.

## 16.2 Mixing 32-Bit and 16-Bit Operations

The 80386 has two instruction prefixes that allow mixing of 32-bit and 16-bit operations within one segment:

- The operand-size prefix (66H)
- The address-size prefix (67H)

These prefixes reverse the default size selected by the D-bit. For example, the processor can interpret the word-move instruction MOV mem, reg in any of four ways:

- In a USE32 segment:

  1. Normally moves 32 bits from a 32-bit register to a 32-bit effective address in memory.

  2. If preceded by an operand-size prefix, moves 16 bits from a 16-bit register to 32-bit effective address in memory.

  3. If preceded by an address-size prefix, moves 32 bits from a 32-bit register to a16-bit effective address in memory.

  4. If preceded by both an address-size prefix and an operand-size prefix, moves 16 bits from a 16-bit register to a 16-bit effective address in memory.

- In a USE16 segment:

  1. Normally moves 16 bits from a 16-bit register to a 16-bit effective address in memory.

  2. If preceded by an operand-size prefix, moves 32 bits from a 32-bit register to 16-bit effective address in memory.

  3. If preceded by an address-size prefix, moves 16 bits from a 16-bit register to a32-bit effective address in memory.

  4. If preceded by both an address-size prefix and an operand-size prefix, moves 32 bits from a 32-bit register to a 32-bit effective address in memory.

These examples illustrate that any instruction can generate any combination of operand size and address size regardless of whether the instruction is in a USE16 or USE32 segment. The choice of the USE16 or USE32 attribute for a code segment is based upon these criteria:

1. The need to address instructions or data in segments that are larger than 64 Kilobytes.

2. The predominant size of operands.

3. The addressing modes desired. (Refer to Chapter 17 for an explanation of the additional addressing modes that are available when 32-bit addressing is used.)

Choosing a setting of the D-bit that is contrary to the predominant size of operands requires the generation of an excessive number of operand-size prefixes.

## 16.3 Sharing Data Segments Among Mixed Code Segments

Because the choice of operand size and address size is defined in code segments and their descriptors, data segments can be shared freely among

both USE16 and USE32 code segments. The only limitation is the one imposed by pointers with 16-bit offsets, which can only point to the first 64 Kilobytes of a segment. When a data segment that contains more than 64 Kilobytes is to be shared among USE32 and USE16 segments, the data that is to be accessed by the USE16 segments must be located within the first 64 Kilobytes.

A stack that spans addresses less than 64K can be shared by both USE16 and USE32 code segments. This class of stacks includes:

- Stacks in expand-up segments with G=0 and B=0.
- Stacks in expand-down segments with G=0 and B=0.
- Stacks in expand-up segments with G=1 and B=0, in which the stack is contained completely within the lower 64 Kilobytes. (Offsets greater than 64K can be used for data, other than the stack, that is not shared.)

The B-bit of a stack segment cannot, in general, be used to change the size of stack used by a USE16 code segment. The size of stack pointer used by the processor for implicit stack references is controlled by the B-bit of the data-segment descriptor for the stack. Implicit references are those caused by interrupts, exceptions, and instructions such as PUSH, POP, CALL, and RET. One might be tempted, therefore, to try to increase beyond 64K the size of the stack used by 16-bit code simply by supplying a larger stack segment with the B-bit set. However, the B-bit does not control explicit stack references, such as accesses to parameters or local variables. A USE16 code segment can utilize a "big" stack only if the code is modified so that all explicit references to the stack are preceded by the address-size prefix, causing those references to use 32-bit addressing.

In big, expand-down segments (B=1, G=1, and E=1), all offsets are greater than 64K, therefore USE16 code cannot utilize such a stack segment unless the code segment is modified to employ 32-bit addressing. (Refer to Chapter 6 for a review of the B, G, and E bits.)

## 16.4 Transferring Control Among Mixed Code Segments

When transferring control among procedures in USE16 and USE32 code segments, programmers must be aware of three points:

- Addressing limitations imposed by pointers with 16-bit offsets.
- Matching of operand-size attribute in effect for the CALL/RET pair and theInterrupt/IRET pair so as to manage the stack correctly.
- Translation of parameters, especially pointer parameters.

Clearly, 16-bit effective addresses cannot be used to address data or code located beyond 64K in a 32-bit segment, nor can large 32-bit parameters be squeezed into a 16-bit word; however, except for these obvious limits, most interfacing problems between 16-bit and 32-bit modules can be solved. Some solutions involve inserting interface procedures between the procedures in question.

### 16.4.1 Size of Code-Segment Pointer

For control-transfer instructions that use a pointer to identify the next instruction (i.e., those that do not use gates), the size of the offset portion of the pointer is determined by the operand-size attribute. The implications of the use of two different sizes of code-segment pointer are:

- JMP, CALL, or RET from 32-bit segment to 16-bit segment is always possible using a 32-bit operand size.

- JMP, CALL, or RET from 16-bit segment using a 16-bit operand size cannot address the target in a 32-bit segment if the address of the target is greater than 64K.

An interface procedure can enable transfers from USE16 segments to 32-bit addresses beyond 64K without requiring modifications any more extensive than relinking or rebinding the old programs. The requirements for such an interface procedure are discussed later in this chapter.

### 16.4.2 Stack Management for Control Transfers

Because stack management is different for 16-bit CALL/RET than for 32-bit CALL/RET, the operand size of RET must match that of CALL. (Refer to Figure 16-1.) A 16-bit CALL pushes the 16-bit IP and (for calls between privilege levels) the 16-bit SP register. The corresponding RET must also use a 16-bit operand size to POP these 16-bit values from the stack into the 16-bit registers. A 32-bit CALL pushes the 32-bit EIP and (for interlevel calls) the 32-bit ESP register. The corresponding RET must also use a 32-bit operand size to POP these 32-bit values from the stack into the 32-bit registers. If the two halves of a CALL/RET pair do not have matching operand sizes, the stack will not be managed correctly and the values of the instruction pointer and stack pointer will not be restored to correct values.

When the CALL and its corresponding RET are in segments that have D-bits with the same values (i.e., both have 32-bit defaults or both have 16-bit defaults), there is no problem. When the CALL and its corresponding RET are in segments that have different D-bit values, however, programmers (or program development software) must ensure that the CALL and RET match.

There are three ways to cause a 16-bit procedure to execute a 32-bit call:

1. Use a 16-bit call to a 32-bit interface procedure that then uses a 32-bit call to invoke the intended target.

2. Bind the 16-bit call to a 32-bit call gate.

3. Modify the 16-bit procedure, inserting an operand-size prefix before the call, thereby changing it to a 32-bit call.

Likewise, there are three ways to cause a 32-bit procedure to execute a 16-bit call:

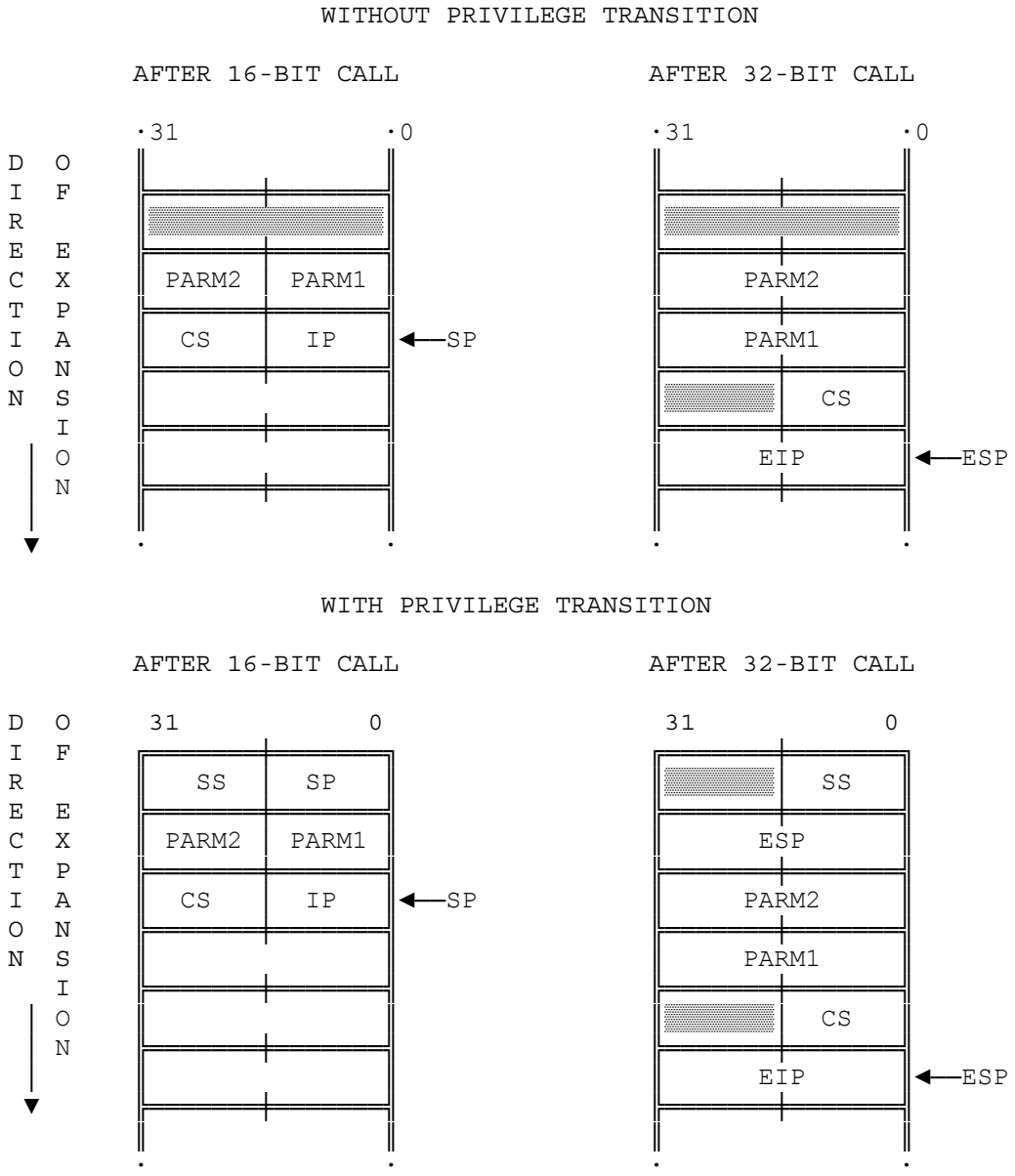
1. Use a 32-bit call to a 32-bit interface procedure that then uses a 16-bit call to invoke the intended target.

2. Bind the 32-bit call to a 16-bit call gate.

3. Modify the 32-bit procedure, inserting an operand-size prefix before the call, thereby changing it to a 16-bit call. (Be certain that the return offset does not exceed 64K.)

Programmers can utilize any of the preceding methods to make a CALL in a USE16 segment match the corresponding RET in a USE32 segment, or to make a CALL in a USE32 segment match the corresponding RET in a USE16 segment.

**Figure 16-1. Stack after Far 16-Bit and 32-Bit Calls**



#### 16.4.2.1 Controlling the Operand-Size for a Call

When the selector of the pointer referenced by a CALL instruction selects a segment descriptor, the operand-size attribute in effect for the CALL instruction is determined by the D-bit in the segment descriptor and by any operand-size instruction prefix.

When the selector of the pointer referenced by a CALL instruction selects a gate descriptor, the type of call is determined by the type of call gate. A call via an 80286 call gate (descriptor type 4) always has a 16-bit operand-size attribute; a call via an 80386 call gate (descriptor type 12) always has a 32-bit operand-size attribute. The offset of the target procedure is taken from the gate descriptor; therefore, even a 16-bit procedure can call a procedure that is located more than 64 kilobytes from the base of a 32-bit segment, because a 32-bit call gate contains a 32-bit target offset.

An unmodified 16-bit code segment that has run successfully on an 8086 or real-mode 80286 will always have a D-bit of zero and will not use operand-size override prefixes; therefore, it will always execute 16-bit versions of CALL. The only modification needed to make a16-bit procedure effect a 32-bit call is to relink the call to an 80386 call gate.

#### 16.4.2.2 Changing Size of Call

When adding 32-bit gates to 16-bit procedures, it is important to consider the number of parameters. The count field of the gate descriptor specifies the size of the parameter string to copy from the current stack to the stack of the more privileged procedure. The count field of a 16-bit gate specifies the number of words to be copied, whereas the count field of a 32-bit gate specifies the number of doublewords to be copied; therefore, the 16-bit procedure must use an even number of words as parameters.

### 16.4.3 Interrupt Control Transfers

With a control transfer due to an interrupt or exception, a gate is always involved. The operand-size attribute for the interrupt is determined by the type of IDT gate.

A 386 interrupt or trap gate (descriptor type 14 or 15) to a 32-bit interrupt procedure can be used to interrupt either 32-bit or 16-bit procedures. However, it is not generally feasible to permit an interrupt or exception to invoke a 16-bit handler procedure when 32-bit code is executing, because a 16-bit interrupt procedure has a return offset of only 16-bits on its stack. If the 32-bit procedure is executing at an address greater than 64K, the 16-bit interrupt procedure cannot return correctly.

### 16.4.4 Parameter Translation

When segment offsets or pointers (which contain segment offsets) are passed as parameters between 16-bit and 32-bit procedures, some translation is required. Clearly, if a 32-bit procedure passes a pointer to data located beyond 64K to a 16-bit procedure, the 16-bit procedure cannot utilize it. Beyond this natural limitation, an interface procedure can perform any format conversion between 32-bit and 16-bit pointers that may be needed.

Parameters passed by value between 32-bit and 16-bit code may also require translation between 32-bit and 16-bit formats. Such translation requirements are application dependent. Systems designers should take care to limit the range of values passed so that such translations are possible.

### 16.4.5 The Interface Procedure

Interposing an interface procedure between 32-bit and 16-bit procedures can be the solution to any of several interface requirements:

- Allowing procedures in 16-bit segments to transfer control to instructions located beyond 64K in 32-bit segments.
- Matching of operand size for CALL/RET.
- Parameter translation.

Interface procedures between USE32 and USE16 segments can be constructed with these properties:

- The procedures reside in a code segment whose D-bit is set, indicating a default operand size of 32-bits.
- All entry points that may be called by 16-bit procedures have offsets that are actually less than 64K.
- All points to which called 16-bit procedures may return also lie within 64K.

The interface procedures do little more than call corresponding procedures in other segments. There may be two kinds of procedures:

- Those that are called by 16-bit procedures and call 32-bit procedures. These interface procedures are called by 16-bit CALLs and use the operand-size prefix before RET instructions to cause a 16-bit RET. CALLs to 32-bit segments are 32-bit calls (by default, because the D-bit is set), and the 32-bit code returns with 32-bit RET instructions.
- Those that are called by 32-bit procedures and call 16-bit procedures. These interface procedures are called by 32-bit CALL instructions, and return with 32-bit RET instructions (by default, because the D-bit is set). CALLs to 16-bit procedures use the operand-size prefix; procedures in the 16-bit code return with 16-bit RET instructions.

# PART IV INSTRUCTION SET

# Chapter 17 80386 Instruction Set

This chapter presents instructions for the 80386 in alphabetical order. For each instruction, the forms are given for each operand combination, including object code produced, operands required, execution time, and a description. For each instruction, there is an operational description and a summary of exceptions generated.

## 17.1 Operand-Size and Address-Size Attributes

When executing an instruction, the 80386 can address memory using either 16 or 32-bit addresses. Consequently, each instruction that uses memory addresses has associated with it an address-size attribute of either 16 or 32 bits. 16-bit addresses imply both the use of a 16-bit displacement in the instruction and the generation of a 16-bit address offset (segment relative address) as the result of the effective address calculation. 32-bit addresses imply the use of a 32-bit displacement and the generation of a 32-bit address offset. Similarly, an instruction that accesses words (16 bits) or doublewords (32 bits) has an operand-size attribute of either 16 or 32 bits.

The attributes are determined by a combination of defaults, instruction prefixes, and (for programs executing in protected mode) size-specification bits in segment descriptors.

### 17.1.1 Default Segment Attribute

For programs executed in protected mode, the D-bit in executable-segment descriptors determines the default attribute for both address size and operand size. These default attributes apply to the execution of all instructions in the segment. A value of zero in the D-bit sets the default address size and operand size to 16 bits; a value of one, to 32 bits.

Programs that execute in real mode or virtual-8086 mode have 16-bit addresses and operands by default.

### 17.1.2 Operand-Size and Address-Size Instruction Prefixes

The internal encoding of an instruction can include two byte-long prefixes: the address-size prefix, 67H, and the operand-size prefix, 66H. (A later section, "Instruction Format," shows the position of the prefixes in an instruction's encoding.) These prefixes override the default segment attributes for the instruction that follows. Table 17-1 shows the effect of each possible combination of defaults and overrides.

### 17.1.3 Address-Size Attribute for Stack

Instructions that use the stack implicitly (for example: POP EAX also have a stack address-size attribute of either 16 or 32 bits. Instructions with a stack address-size attribute of 16 use the 16-bit SP stack pointer register; instructions with a stack address-size attribute of 32 bits use the 32-bit ESP register to form the address of the top of the stack.

The stack address-size attribute is controlled by the B-bit of the data-segment descriptor in the SS register. A value of zero in the B-bit selects a stack address-size attribute of 16; a value of one selects a stack address-size attribute of 32.

**Table 17-1. Effective Size Attributes**

| Segment Default D = ... | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|
| Operand-Size Prefix 66H | N | N | Y | Y | N | N | Y | Y |
| Address-Size Prefix 67H | N | Y | N | Y | N | Y | N | Y |
| Effective Operand Size | 16 | 16 | 32 | 32 | 32 | 32 | 16 | 16 |
| Effective Address Size | 16 | 32 | 16 | 32 | 32 | 16 | 32 | 16 |

Y = Yes, this instruction prefix is present
N = No, this instruction prefix is not present

## 17.2 Instruction Format

All instruction encodings are subsets of the general instruction format shown in Figure 17-1. Instructions consist of optional instruction prefixes, one or two primary opcode bytes, possibly an address specifier consisting of the ModR/M byte and the SIB (Scale Index Base) byte, a displacement, if required, and an immediate data field, if required.

Smaller encoding fields can be defined within the primary opcode or opcodes. These fields define the direction of the operation, the size of the displacements, the register encoding, or sign extension; encoding fields vary depending on the class of operation.

Most instructions that can refer to an operand in memory have an addressing form byte following the primary opcode byte(s). This byte, called the ModR/M byte, specifies the address form to be used. Certain encodings of the ModR/M byte indicate a second addressing byte, the SIB (Scale Index Base) byte, which follows the ModR/M byte and is required to fully specify the addressing form.

Addressing forms can include a displacement immediately following either the ModR/M or SIB byte. If a displacement is present, it can be 8-, 16- or 32-bits.

If the instruction specifies an immediate operand, the immediate operand always follows any displacement bytes. The immediate operand, if specified, is always the last field of the instruction.

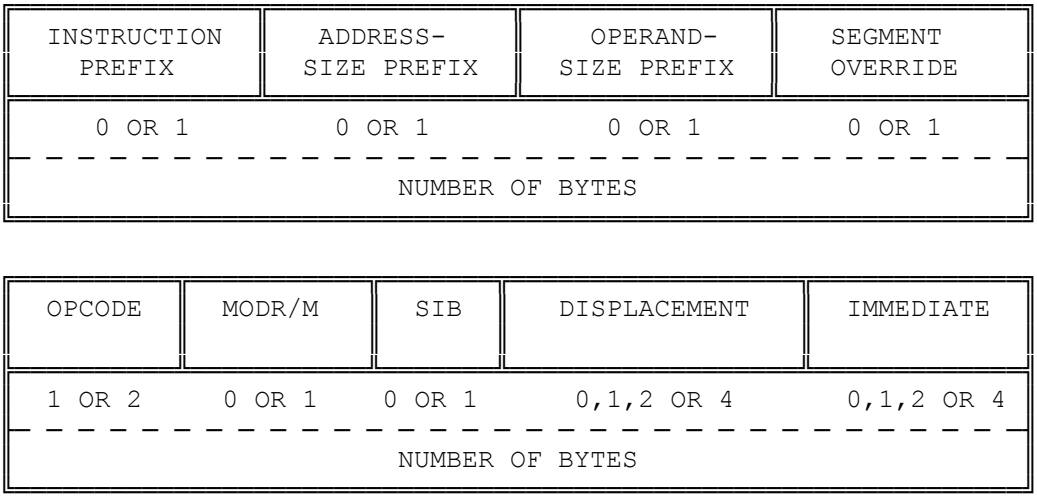
The following are the allowable instruction prefix codes:

| | |
|---|---|
| F3H | REP prefix (used only with string instructions) |
| F3H | REPE/REPZ prefix (used only with string instructions |
| F2H | REPNE/REPNZ prefix (used only with string instructions) |
| F0H | LOCK prefix |

The following are the segment override prefixes:

| | |
|---|---|
| 2EH | CS segment override prefix |
| 36H | SS segment override prefix |
| 3EH | DS segment override prefix |
| 26H | ES segment override prefix |
| 64H | FS segment override prefix |
| 65H | GS segment override prefix |
| 66H | Operand-size override |
| 67H | Address-size override |

**Figure 17-1. 80386 Instruction Format**




### 17.2.1 ModR/M and SIB Bytes

The ModR/M and SIB bytes follow the opcode byte(s) in many of the 80386 instructions. They contain the following information:

- The indexing type or register number to be used in the instruction
- The register to be used, or more information to select the instruction
- The base, index, and scale information

The ModR/M byte contains three fields of information:

- The mod field, which occupies the two most significant bits of the byte, combines with the r/m field to form 32 possible values: eight registers and 24 indexing modes

- The reg field, which occupies the next three bits following the mod field, specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the first (opcode) byte of the instruction.

- The r/m field, which occupies the three least significant bits of the byte, can specify a register as the location of an operand, or can form part of the addressing-mode encoding in combination with the field as described above

The based indexed and scaled indexed forms of 32-bit addressing require the SIB byte. The presence of the SIB byte is indicated by certain encodings of the ModR/M byte. The SIB byte then includes the following fields:

- The ss field, which occupies the two most significant bits of the byte, specifies the scale factor

- The index field, which occupies the next three bits following the ss field and specifies the register number of the index register

- The base field, which occupies the three least significant bits of the byte, specifies the register number of the base register

Figure 17-2 shows the formats of the ModR/M and SIB bytes.

The values and the corresponding addressing forms of the ModR/M and SIB bytes are shown in Tables 17-2, 17-3, and 17-4. The 16-bit addressing forms specified by the ModR/M byte are in Table 17-2. The 32-bit addressing forms specified by ModR/M are in Table 17-3. Table 17-4 shows the 32-bit addressing forms specified by the SIB byte

**Figure 17-2. ModR/M and SIB Byte Formats**

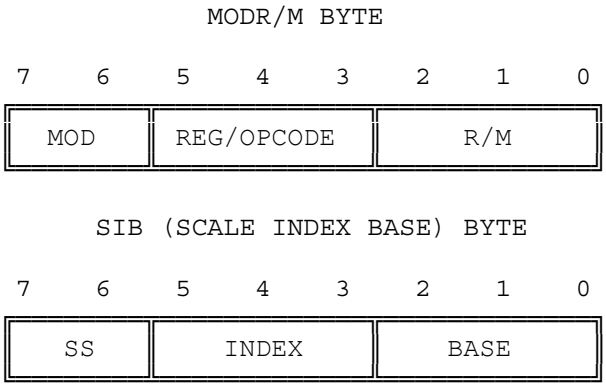


**Table 17-2. 16-Bit Addressing Forms with the ModR/M Byte**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| r8(/r) | | AL | CL | DL | BL | AH | CH | DH | BH |
| r16(/r) | | AX | CX | DX | BX | SP | BP | SI | DI |
| r32(/r) | | EAX | ECX | EDX | EBX | ESP | EBP | ESI | EDI |
| /digit (Opcode) | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| REG = | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| Effective Address | Mod R/M | ModR/M Values in Hexadecimal | | | | | | | |
| [BX + SI] | | 000 | 00 | 08 | 10 | 18 | 20 | 28 | 30 | 38 |
| [BX + DI] | | 001 | 01 | 09 | 11 | 19 | 21 | 29 | 31 | 39 |
| [BP + SI] | | 010 | 02 | 0A | 12 | 1A | 22 | 2A | 32 | 3A |
| [BP + DI] | | 011 | 03 | 0B | 13 | 1B | 23 | 2B | 33 | 3B |
| [SI] | 00 | 100 | 04 | 0C | 14 | 1C | 24 | 2C | 34 | 3C |
| [DI] | | 101 | 05 | 0D | 15 | 1D | 25 | 2D | 35 | 3D |
| disp16 | | 110 | 06 | 0E | 16 | 1E | 26 | 2E | 36 | 3E |
| [BX] | | 111 | 07 | 0F | 17 | 1F | 27 | 2F | 37 | 3F |
| [BX+SI]+disp8 | | 000 | 40 | 48 | 50 | 58 | 60 | 68 | 70 | 78 |
| [BX+DI]+disp8 | | 001 | 41 | 49 | 51 | 59 | 61 | 69 | 71 | 79 |
| [BP+SI]+disp8 | | 010 | 42 | 4A | 52 | 5A | 62 | 6A | 72 | 7A |
| [BP+DI]+disp8 | | 011 | 43 | 4B | 53 | 5B | 63 | 6B | 73 | 7B |
| [SI]+disp8 | 01 | 100 | 44 | 4C | 54 | 5C | 64 | 6C | 74 | 7C |
| [DI]+disp8 | | 101 | 45 | 4D | 55 | 5D | 65 | 6D | 75 | 7D |
| [BP]+disp8 | | 110 | 46 | 4E | 56 | 5E | 66 | 6E | 76 | 7E |
| [BX]+disp8 | | 111 | 47 | 4F | 57 | 5F | 67 | 6F | 77 | 7F |
| [BX+SI]+disp16 | | 000 | 80 | 88 | 90 | 98 | A0 | A8 | B0 | B8 |
| [BX+DI]+disp16 | | 001 | 81 | 89 | 91 | 99 | A1 | A9 | B1 | B9 |
| [BX+SI]+disp16 | | 010 | 82 | 8A | 92 | 9A | A2 | AA | B2 | BA |
| [BX+DI]+disp16 | | 011 | 83 | 8B | 93 | 9B | A3 | AB | B3 | BB |
| [SI]+disp16 | 10 | 100 | 84 | 8C | 94 | 9C | A4 | AC | B4 | BC |
| [DI]+disp16 | | 101 | 85 | 8D | 95 | 9D | A5 | AD | B5 | BD |
| [BP]+disp16 | | 110 | 86 | 8E | 96 | 9E | A6 | AE | B6 | BE |
| [BX]+disp16 | | 111 | 87 | 8F | 97 | 9F | A7 | AF | B7 | BF |
| EAX/AX/AL | | 000 | C0 | C8 | D0 | D8 | E0 | E8 | F0 | F8 |
| ECX/CX/CL | | 001 | C1 | C9 | D1 | D9 | E1 | E9 | F1 | F9 |
| EDX/DX/DL | | 010 | C2 | CA | D2 | DA | E2 | EA | F2 | FA |
| EBX/BX/BL | | 011 | C3 | CB | D3 | DB | E3 | EB | F3 | FB |
| ESP/SP/AH | 11 | 100 | C4 | CC | D4 | DC | E4 | EC | F4 | FC |
| EBP/BP/CH | | 101 | C5 | CD | D5 | DD | E5 | ED | F5 | FD |
| ESI/SI/DH | | 110 | C6 | CE | D6 | DE | E6 | EE | F6 | FE |
| EDI/DI/BH | | 111 | C7 | CF | D7 | DF | E7 | EF | F7 | FF |

NOTES:
disp8 denotes an 8-bit displacement following the ModR/M byte, to be sign-extended and added to the index. disp16 denotes a 16-bit displacement following the ModR/M byte, to be added to the index. Default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.

**Table 17-3. 32-Bit Addressing Forms with the ModR/M Byte**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| r8(/r) | | AL | CL | DL | BL | AH | CH | DH | BH |
| r16(/r) | | AX | CX | DX | BX | SP | BP | SI | DI |
| r32(/r) | | EAX | ECX | EDX | EBX | ESP | EBP | ESI | EDI |
| /digit (Opcode) | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| REG = | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| Effective Address | Mod R/M | ModR/M Values in Hexadecimal | | | | | | | |
| [EAX] | 000 | 00 | 08 | 10 | 18 | 20 | 28 | 30 | 38 |
| [ECX] | 001 | 01 | 09 | 11 | 19 | 21 | 29 | 31 | 39 |
| [EDX] | 010 | 02 | 0A | 12 | 1A | 22 | 2A | 32 | 3A |
| [EBX] | 011 | 03 | 0B | 13 | 1B | 23 | 2B | 33 | 3B |
| [--] [--] | 00 100 | 04 | 0C | 14 | 1C | 24 | 2C | 34 | 3C |
| disp32 | 101 | 05 | 0D | 15 | 1D | 25 | 2D | 35 | 3D |
| [ESI] | 110 | 06 | 0E | 16 | 1E | 26 | 2E | 36 | 3E |
| [EDI] | 111 | 07 | 0F | 17 | 1F | 27 | 2F | 37 | 3F |
| disp8[EAX] | 000 | 40 | 48 | 50 | 58 | 60 | 68 | 70 | 78 |
| disp8[ECX] | 001 | 41 | 49 | 51 | 59 | 61 | 69 | 71 | 79 |
| disp8[EDX] | 010 | 42 | 4A | 52 | 5A | 62 | 6A | 72 | 7A |
| disp8[EPX]; | 011 | 43 | 4B | 53 | 5B | 63 | 6B | 73 | 7B |
| disp8[--] [--] | 01 100 | 44 | 4C | 54 | 5C | 64 | 6C | 74 | 7C |
| disp8[ebp] | 101 | 45 | 4D | 55 | 5D | 65 | 6D | 75 | 7D |
| disp8[ESI] | 110 | 46 | 4E | 56 | 5E | 66 | 6E | 76 | 7E |
| disp8[EDI] | 111 | 47 | 4F | 57 | 5F | 67 | 6F | 77 | 7F |
| disp32[EAX] | 000 | 80 | 88 | 90 | 98 | A0 | A8 | B0 | B8 |
| disp32[ECX] | 001 | 81 | 89 | 91 | 99 | A1 | A9 | B1 | B9 |
| disp32[EDX] | 010 | 82 | 8A | 92 | 9A | A2 | AA | B2 | BA |
| disp32[EBX] | 011 | 83 | 8B | 93 | 9B | A3 | AB | B3 | BB |
| disp32[--] [--] | 10 100 | 84 | 8C | 94 | 9C | A4 | AC | B4 | BC |
| disp32[EBP] | 101 | 85 | 8D | 95 | 9D | A5 | AD | B5 | BD |
| disp32[ESI] | 110 | 86 | 8E | 96 | 9E | A6 | AE | B6 | BE |
| disp32[EDI] | 111 | 87 | 8F | 97 | 9F | A7 | AF | B7 | BF |
| EAX/AX/AL | 000 | C0 | C8 | D0 | D8 | E0 | E8 | F0 | F8 |
| ECX/CX/CL | 001 | C1 | C9 | D1 | D9 | E1 | E9 | F1 | F9 |
| EDX/DX/DL | 010 | C2 | CA | D2 | DA | E2 | EA | F2 | FA |
| EBX/BX/BL | 011 | C3 | CB | D3 | DB | E3 | EB | F3 | FB |
| ESP/SP/AH | 11 100 | C4 | CC | D4 | DC | E4 | EC | F4 | FC |
| EBP/BP/CH | 101 | C5 | CD | D5 | DD | E5 | ED | F5 | FD |
| ESI/SI/DH | 110 | C6 | CE | D6 | DE | E6 | EE | F6 | FE |
| EDI/DI/BH | 111 | C7 | CF | D7 | DF | E7 | EF | F7 | FF |

NOTES:
[--] [--] means a SIB follows the ModR/M byte. disp8 denotes an 8-bit displacement following the SIB byte, to be sign-extended and added to the index. disp32 denotes a 32-bit displacement following the ModR/M byte, to be added to the index.

**Table 17-4. 32-Bit Addressing Forms with the SIB Byte**

| r32 | | EAX | ECX | EDX | EBX | ESP | EBP | ESI | EDI |
|---|---|---|---|---|---|---|---|---|---|
| Base = | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Base = | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| Scaled Index | SS Index | ModR/M Values in Hexadecimal | | | | | | | |
| [EAX] | 000 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| [ECX] | 001 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
| [EDX] | 010 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| [EBX] | 011 | 18 | 19 | 1A | 1B | 1C | 1D | 1E | 1F |
| none | 00 100 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| [EBP] | 101 | 28 | 29 | 2A | 2B | 2C | 2D | 2E | 2F |
| [ESI] | 110 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| [EDI] | 111 | 38 | 39 | 3A | 3B | 3C | 3D | 3E | 3F |
| [EAX*2] | 000 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| [ECX*2] | 001 | 48 | 49 | 4A | 4B | 4C | 4D | 4E | 4F |
| [ECX*2] | 010 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 |
| [EBX*2] | 011 | 58 | 59 | 5A | 5B | 5C | 5D | 5E | 5F |
| none | 01 100 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 |
| [EBP*2] | 101 | 68 | 69 | 6A | 6B | 6C | 6D | 6E | 6F |
| [ESI*2] | 110 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |
| [EDI*2] | 111 | 78 | 79 | 7A | 7B | 7C | 7D | 7E | 7F |
| [EAX*4] | 000 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 |
| [ECX*4] | 001 | 88 | 89 | 8A | 8B | 8C | 8D | 8E | 8F |
| [EDX*4] | 010 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 |
| [EBX*4] | 011 | 98 | 89 | 9A | 9B | 9C | 9D | 9E | 9F |
| none | 10 100 | A0 | A1 | A2 | A3 | A4 | A5 | A6 | A7 |
| [EBP*4] | 101 | A8 | A9 | AA | AB | AC | AD | AE | AF |
| [ESI*4] | 110 | B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 |
| [EDI*4] | 111 | B8 | B9 | BA | BB | BC | BD | BE | BF |
| [EAX*8] | 000 | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| [ECX*8] | 001 | C8 | C9 | CA | CB | CC | CD | CE | CF |
| [EDX*8] | 010 | D0 | D1 | D2 | D3 | D4 | D5 | D6 | D7 |
| [EBX*8] | 011 | D8 | D9 | DA | DB | DC | DD | DE | DF |
| none | 11 100 | E0 | E1 | E2 | E3 | E4 | E5 | E6 | E7 |
| [EBP*8] | 101 | E8 | E9 | EA | EB | EC | ED | EE | EF |
| [ESI*8] | 110 | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 |
| [EDI*8] | 111 | F8 | F9 | FA | FB | FC | FD | FE | FF |

NOTES:
[*] means a disp32 with no base if MOD is 00, [ESP] otherwise. This provides the following addressing modes:

```
disp32[index]          (MOD=00)
disp8[EBP][index]      (MOD=01)
disp32[EBP][index]     (MOD=10)
```

### 17.2.2 How to Read the Instruction Set Pages

The following is an example of the format used for each 80386 instruction description in this chapter:

CMC —— Complement Carry Flag

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F5 | CMC | 2 | Complement carry flag |

The above table is followed by paragraphs labelled "Operation," "Description," "Flags Affected," "Protected Mode Exceptions," "Real Address Mode Exceptions," and, optionally, "Notes." The following sections explain the notational conventions and abbreviations used in these paragraphs of the instruction descriptions.

#### 17.2.2.1 Opcode

The "Opcode" column gives the complete object code produced for each form of the instruction. When possible, the codes are given as hexadecimal bytes, in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

/digit: (digit is between 0 and 7) indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.

/r: indicates that the ModR/M byte of the instruction contains both a register operand and an r/m operand.

cb, cw, cd, cp: a 1-byte (cb), 2-byte (cw), 4-byte (cd) or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.

ib, iw, id: a 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and doublewords are given with the low-order byte first.

+rb, +rw, +rd: a register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The codes are——

| rb | rw | rd |
|---|---|---|
| AL = 0 | AX = 0 | EAX = 0 |
| CL = 1 | CX = 1 | ECX = 1 |
| DL = 2 | DX = 2 | EDX = 2 |
| BL = 3 | BX = 3 | EBX = 3 |
| AH = 4 | SP = 4 | ESP = 4 |
| CH = 5 | BP = 5 | EBP = 5 |
| DH = 6 | SI = 6 | ESI = 6 |
| BH = 7 | DI = 7 | EDI = 7 |

#### 17.2.2.2 Instruction

The "Instruction" column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

rel8: a relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.

rel16, rel32: a relative address within the same code segment as the instruction assembled. rel16 applies to instructions with an operand-size attribute of 16 bits; rel32 applies to instructions with an operand-size attribute of 32 bits.

ptr16:16, ptr16:32: a FAR pointer, typically in a code segment different from that of the instruction. The notation 16:16 indicates that the value of the pointer has two parts. The value to the right of the colon is a 16-bit selector or value destined for the code segment register. The value to the left corresponds to the offset within the destination segment. ptr16:16 is used when the instruction's operand-size attribute is 16 bits; ptr16:32 is used with the 32-bit attribute.

r8: one of the byte registers AL, CL, DL, BL, AH, CH, DH, or BH.

r16: one of the word registers AX, CX, DX, BX, SP, BP, SI, or DI.

r32: one of the doubleword registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.

imm8: an immediate byte value. imm8 is a signed number between -128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.

imm16: an immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between -32768 and +32767 inclusive.

imm32: an immediate doubleword value used for instructions whose operand-size attribute is 32-bits. It allows the use of a number between +2147483647 and -2147483648.

r/m8: a one-byte operand that is either the contents of a byte register (AL, BL, CL, DL, AH, BH, CH, DH), or a byte from memory.

r/m16: a word register or memory operand used for instructions whose operand-size attribute is 16 bits. The word registers are: AX, BX, CX, DX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation.

r/m32: a doubleword register or memory operand used for instructions whose operand-size attribute is 32-bits. The doubleword registers are: EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation.

m8: a memory byte addressed by DS:SI or ES:DI (used only by string instructions).

m16: a memory word addressed by DS:SI or ES:DI (used only by string instructions).

m32: a memory doubleword addressed by DS:SI or ES:DI (used only by string instructions).

m16:16, M16:32: a memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.

m16 & 32, m16 & 16, m32 & 32: a memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. m16 & 16 and m32 & 32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. m16 & 32 is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding Global and Interrupt Descriptor Table Registers.

moffs8, moffs16, moffs32: (memory offset) a simple memory variable of type BYTE, WORD, or DWORD used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.

Sreg: a segment register. The segment register bit assignments are ES=0, CS=1, SS=2, DS=3, FS=4, and GS=5.

#### 17.2.2.3 Clocks

The "Clocks" column gives the number of clock cycles the instruction takes to execute. The clock count calculations makes the following assumptions:

- The instruction has been prefetched and decoded and is ready for execution.
- Bus cycles do not require wait states.
- There are no local bus HOLD requests delaying processor access to the bus.
- No exceptions are detected during instruction execution.
- Memory operands are aligned.

Clock counts for instructions that have an r/m (register or memory) operand are separated by a slash. The count to the left is used for a register operand; the count to the right is used for a memory operand.

The following symbols are used in the clock count specifications:

- n, which represents a number of repetitions.
- m, which represents the number of components in the next instruction executed, where the entire displacement (if any) counts as one component, the entire immediate data (if any) counts as one component, and every other byte of the instruction and prefix(es) each counts as one component.
- pm=, a clock count that applies when the instruction executes in Protected Mode. pm= is not given when the clock counts are the same for Protected and Real Address Modes.

When an exception occurs during the execution of an instruction and the exception handler is in another task, the instruction execution time is increased by the number of clocks to effect a task switch. This parameter depends on several factors:

- The type of TSS used to represent the current task (386 TSS or 286 TSS).
- The type of TSS used to represent the new task.
- Whether the current task is in V86 mode.
- Whether the new task is in V86 mode.

Table 17-5 summarizes the task switch times for exceptions.

**Table 17-5. Task Switch Times for Exceptions**

| | | New Task | |
|---|---|---|---|
| Old Task | | 386 TSS VM = 0 | 286 TSS |
| 386 TSS | VM = 0 | 309 | 282 |
| 386 TSS | VM = 1 | 314 | 231 |
| 286 TSS | | 307 | 282 |

### 17.2.2.4 Description

The "Description" column following the "Clocks" column briefly explains the various forms of the instruction. The "Operation" and "Description" sections contain more details of the instruction's operation.

### 17.2.2.5 Operation

The "Operation" section contains an algorithmic description of the instruction which uses a notation similar to the Algol or Pascal language. The algorithms are composed of the following elements:

Comments are enclosed within the symbol pairs "(*" and "*)".

Compound statements are enclosed between the keywords of the "if" statement (IF, THEN, ELSE, FI) or of the "do" statement (DO, OD), or of the "case" statement (CASE ... OF, ESAC).

A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to SI's default segment (DS) or overridden segment.

Brackets also used for memory operands, where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the contents of the source operand is a segment-relative offset.

A ← B; indicates that the value of B is assigned to A.

The symbols =, ≠, ≥, and ≤ are relational operators used to compare two values, meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.

The following identifiers are used in the algorithmic descriptions:

- OperandSize represents the operand-size attribute of the instruction, which is either 16 or 32 bits. AddressSize represents the address-size attribute, which is either 16 or 32 bits. For example,

```
IF instruction = CMPSW
THEN OperandSize ← 16;
ELSE
   IF instruction = CMPSD
   THEN OperandSize ← 32;
   FI;
FI;
```

indicates that the operand-size attribute depends on the form of the CMPS instruction used. Refer to the explanation of address-size and operand-size attributes at the beginning of this chapter for general guidelines on how these attributes are determined.

- StackAddrSize represents the stack address-size attribute associated with the instruction, which has a value of 16 or 32 bits, as explained earlier in the chapter.

- SRC represents the source operand. When there are two operands, SRC is the one on the right.

- DEST represents the destination operand. When there are two operands, DEST is the one on the left.

- LeftSRC, RightSRC distinguishes between two operands when both are source operands.

- eSP represents either the SP register or the ESP register depending on the setting of the B-bit for the current stack segment.

The following functions are used in the algorithmic descriptions:

- Truncate to 16 bits(value) reduces the size of the value to fit in 16 bits by discarding the uppermost bits as needed.

- Addr(operand) returns the effective address of the operand (the result of the effective address calculation prior to adding the segment base).

- ZeroExtend(value) returns a value zero-extended to the operand-size attribute of the instruction. For example, if OperandSize = 32, ZeroExtend of a byte value of -10 converts the byte from F6H to doubleword with hexadecimal value 000000F6H. If the value passed to ZeroExtend and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.

- SignExtend(value) returns a value sign-extended to the operand-size attribute of the instruction. For example, if OperandSize = 32, SignExtend of a byte containing the value -10 converts the byte from F6H to a doubleword with hexadecimal value FFFFFFF6H. If the value passed to SignExtend and the operand-size attribute are the same size, SignExtend returns the value unaltered.

- Push(value) pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. The action of Push is as follows:

```
IF StackAddrSize = 16
THEN
   IF OperandSize = 16
   THEN
      SP ← SP - 2;
      SS:[SP] ← value; (* 2 bytes assigned starting at
                          byte address in SP *)
   ELSE (* OperandSize = 32 *)
      SP ← SP - 4;
      SS:[SP] ← value; (* 4 bytes assigned starting at
                          byte address in SP *)
   FI;
ELSE (* StackAddrSize = 32 *)
   IF OperandSize = 16
   THEN
      ESP ← ESP - 2;
      SS:[ESP] ← value; (* 2 bytes assigned starting at
                           byte address in ESP*)
   ELSE (* OperandSize = 32 *)
      ESP ← ESP - 4;
      SS:[ESP] ← value; (* 4 bytes assigned starting at
                           byte address in ESP*)
```

```
   FI;
FI;
```

- Pop(value) removes the value from the top of the stack and returns it. The statement EAX ← Pop( ); assigns to EAX the 32-bit value that Pop took from the top of the stack. Pop will return either a word or a doubleword depending on the operand-size attribute. The action of Pop is as follows:

```
IF StackAddrSize = 16
THEN
   IF OperandSize = 16
   THEN
      ret val ← SS:[SP]; (* 2-byte value *)
      SP ← SP + 2;
   ELSE (* OperandSize = 32 *)
      ret val ← SS:[SP]; (* 4-byte value *)
      SP ← SP + 4;
   FI;
ELSE (* StackAddrSize = 32 *)
   IF OperandSize = 16
   THEN
      ret val ← SS:[ESP]; (* 2 bytes value *)
      ESP ← ESP + 2;
   ELSE (* OperandSize = 32 *)
      ret val ← SS:[ESP]; (* 4 bytes value *)
      ESP ← ESP + 4;
   FI;
FI;
RETURN(ret val); (*returns a word or doubleword*)
```

- Bit[BitBase, BitOffset] returns the address of a bit within a bit string, which is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. In memory, the two bytes of a word are stored with the low-order byte at the lower address.

  If the base operand is a register, the offset can be in the range 0..31. This offset addresses a bit within the indicated register. An example, "BIT[EAX, 21]," is illustrated in Figure 17-3.

  If BitBase is a memory address, BitOffset can range from -2 gigabits to 2 gigabits. The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)), where DIV is signed division with rounding towards negative infinity, and MOD returns a positive number. This is illustrated in Figure 17-4.

- I-O-Permission(I-O-Address, width) returns TRUE or FALSE depending on the I/O permission bitmap and other factors. This function is defined as follows:

```
IF TSS type is 286 THEN RETURN FALSE; FI;
Ptr ← [TSS + 66]; (* fetch bitmap pointer *)
BitStringAddr ← SHR (I-O-Address, 3) + Ptr;
MaskShift ← I-O-Address AND 7;
CASE width OF:
      BYTE: nBitMask ← 1;
```

```
        WORD: nBitMask ← 3;
        DWORD: nBitMask ← 15;
ESAC;
mask ← SHL (nBitMask, MaskShift);
CheckString ← [BitStringAddr] AND mask;
IF CheckString = 0
THEN RETURN (TRUE);
ELSE RETURN (FALSE);
FI;
```

- Switch-Tasks is the task switching function described in Chapter 7.

### 17.2.2.6 Description

The "Description" section contains further explanation of the instruction's operation.

**Figure 17-3. Bit Offset for BIT[EAX, 21]**

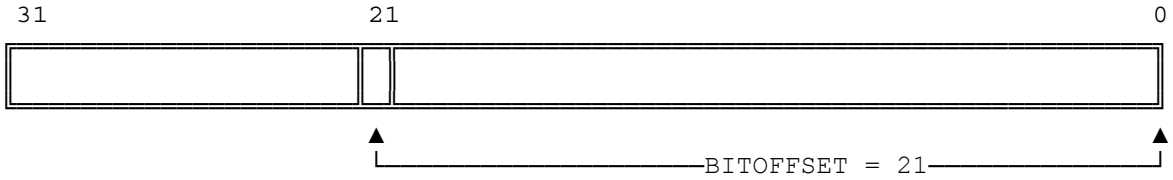
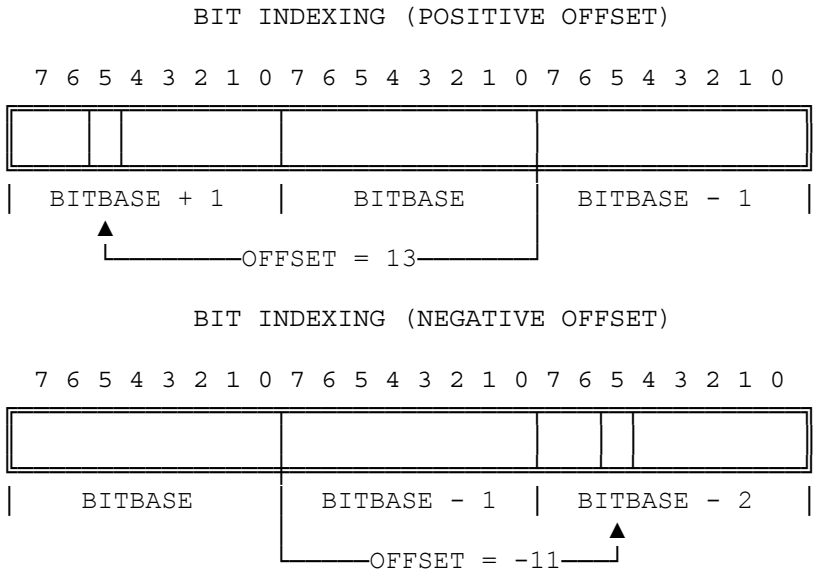



**Figure 17-4. Memory Bit Indexing**



#### 17.2.2.7 Flags Affected

The "Flags Affected" section lists the flags that are affected by the instruction, as follows:

- If a flag is always cleared or always set by the instruction, the value is given (0 or 1) after the flag name. Arithmetic and logical instructions usually assign values to the status flags in the uniform manner described in Appendix C. Nonconventional assignments are described in the "Operation" section.

- The values of flags listed as "undefined" may be changed by the instruction in an indeterminate manner.

All flags not listed are unchanged by the instruction.

#### 17.2.2.8 Protected Mode Exceptions

This section lists the exceptions that can occur when the instruction is executed in 80386 Protected Mode. The exception names are a pound sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. Table 17-6 associates each two-letter name with the corresponding interrupt number.

Chapter 9 describes the exceptions and the 80386 state upon entry to the exception.

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

**Table 17-6. 80386 Exceptions**

| Mnemonic | Interrupt | Description |
|---|---|---|
| #UD | 6 | Invalid opcode |
| #NM | 7 | Coprocessor not available |
| #DF | 8 | Double fault |
| #TS | 10 | Invalid TSS |
| #NP | 11 | Segment or gate not present |
| #SS | 12 | Stack fault |
| #GP | 13 | General protection fault |
| #PF | 14 | Page fault |
| #MF | 16 | Math (coprocessor) fault |

#### 17.2.2.9 Real Address Mode Exceptions

Because less error checking is performed by the 80386 in Real Address Mode, this mode has fewer exception conditions. Refer to Chapter 14 for further information on these exceptions.

#### 17.2.2.10 Virtual-8086 Mode Exceptions

Virtual 8086 tasks provide the ability to simulate Virtual 8086 machines. Virtual 8086 Mode exceptions are similar to those for the 8086 processor, but there are some differences. Refer to Chapter 15 for details.

#### 17.2.2.11 Instruction Set Detail

The instruction set is detailed as follows:

## AAA — ASCII Adjust after Addition

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 37 | AAA | 4 | ASCII adjust AL after addition |

### Operation

```
IF ((AL AND 0FH) > 9) OR (AF = 1)
THEN
   AL ← (AL + 6) AND 0FH;
   AH ← AH + 1;
   AF ← 1;
   CF ← 1;
ELSE
   CF ← 0;
   AF ← 0;
FI;
```

### Description

Execute AAA only following an ADD instruction that leaves a byte result in the AL register. The lower nibbles of the operands of the ADD instruction should be in the range 0 through 9 (BCD digits). In this case, AAA adjusts AL to contain the correct decimal digit result. If the addition produced a decimal carry, the AH register is incremented, and the carry and auxiliary carry flags are set to 1. If there was no decimal carry, the carry and auxiliary flags are set to 0 and AH is unchanged. In either case, AL is left with its top nibble set to 0. To convert AL to an ASCII result, follow the AAA instruction with OR AL, 30H.

### Flags Affected

AF and CF as described above; OF, SF, ZF, and PF are undefined

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

## AAD —— ASCII Adjust AX before Division

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| D5 0A | AAD | 19 | ASCII adjust AX before division |

### Operation

```
AL ← AH * 10 + AL;
AH ← 0;
```

### Description

AAD is used to prepare two unpacked BCD digits (the least-significant digit in AL, the most-significant digit in AH) for a division operation that will yield an unpacked result. This is accomplished by setting AL to AL + (10 * AH), and then setting AH to 0. AX is then equal to the binary equivalent of the original unpacked two-digit number.

### Flags Affected

SF, ZF, and PF as described in Appendix C; OF, AF, and CF are undefined

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

## AAM — ASCII Adjust AX after Multiply

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| D4 0A | AAM | 17 | ASCII adjust AX after multiply |

### Operation

```
AH ← AL / 10;
AL ← AL MOD 10;
```

### Description

Execute AAM only after executing a MUL instruction between two unpacked BCD digits that leaves the result in the AX register. Because the result is less than 100, it is contained entirely in the AL register. AAM unpacks the AL result by dividing AL by 10, leaving the quotient (most-significant digit) in AH and the remainder (least-significant digit) in AL.

### Flags Affected

SF, ZF, and PF as described in Appendix C; OF, AF, and CF are undefined

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

## AAS — ASCII Adjust AL after Subtraction

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 3F | AAS | 4 | ASCII adjust AL after subtraction |

### Operation

```
IF (AL AND 0FH) > 9 OR AF = 1
THEN
   AL ← AL - 6;
   AL ← AL AND 0FH;
   AH ← AH - 1;
   AF ← 1;
   CF ← 1;
ELSE
   CF ← 0;
   AF ← 0;
FI;
```

### Description

Execute AAS only after a SUB instruction that leaves the byte result in the AL register. The lower nibbles of the operands of the SUB instruction must have been in the range 0 through 9 (BCD digits). In this case, AAS adjusts AL so it contains the correct decimal digit result. If the subtraction produced a decimal carry, the AH register is decremented, and the carry and auxiliary carry flags are set to 1. If no decimal carry occurred, the carry and auxiliary carry flags are set to 0, and AH is unchanged. In either case, AL is left with its top nibble set to 0. To convert AL to an ASCII result, follow the AAS with OR AL, 30H.

### Flags Affected

AF and CF as described above; OF, SF, ZF, and PF are undefined

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

**ADC —— Add with Carry**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 14 ib | ADC AL,imm8 | 2 | Add with carry immediate byte to AL |
| 15 iw | ADC AX,imm16 | 2 | Add with carry immediate word to AX |
| 15 id | ADC EAX,imm32 | 2 | Add with carry immediate dword to EAX |
| 80 /2 ib | ADC r/m8,imm8 | 2/7 | Add with carry immediate byte to r/m byte |
| 81 /2 iw | ADC r/m16,imm16 | 2/7 | Add with carry immediate word to r/m word |
| 81 /2 id | ADC r/m32,imm32 | 2/7 | Add with CF immediate dword to r/m dword |
| 83 /2 ib | ADC r/m16,imm8 | 2/7 | Add with CF sign-extended immediate byte to r/m word |
| 83 /2 ib | ADC r/m32,imm8 | 2/7 | Add with CF sign-extended immediate byte into r/m dword |
| 10 /r | ADC r/m8,r8 | 2/7 | Add with carry byte register to r/m byte |
| 11 /r | ADC r/m16,r16 | 2/7 | Add with carry word register to r/m word |
| 11 /r | ADC r/m32,r32 | 2/7 | Add with CF dword register to r/m dword |
| 12 /r | ADC r8,r/m8 | 2/6 | Add with carry r/m byte to byte register |
| 13 /r | ADC r16,r/m16 | 2/6 | Add with carry r/m word to word register |
| 13 /r | ADC r32,r/m32 | 2/6 | Add with CF r/m dword to dword register |

Operation

DEST ← DEST + SRC + CF;

Description

ADC performs an integer addition of the two operands DEST and SRC and the carry flag, CF. The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly. ADC is usually executed as part of a multi-byte or multi-word addition operation. When an immediate byte value is added to a word or doubleword operand, the immediate value is first sign-extended to the size of the word or doubleword operand.

Flags Affected

OF, SF, ZF, AF, CF, and PF as described in Appendix C

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) if page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

**ADD —— Add**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 04 ib | ADD AL,imm8 | 2 | Add immediate byte to AL |
| 05 iw | ADD AX,imm16 | 2 | Add immediate word to AX |
| 05 id | ADD EAX,imm32 | 2 | Add immediate dword to EAX |
| 80 /0 ib | ADD r/m8,imm8 | 2/7 | Add immediate byte to r/m byte |
| 81 /0 iw | ADD r/m16,imm16 | 2/7 | Add immediate word to r/m word |
| 81 /0 id | ADD r/m32,imm32 | 2/7 | Add immediate dword to r/m dword |
| 83 /0 ib | ADD r/m16,imm8 | 2/7 | Add sign-extended immediate byte to r/m word |
| 83 /0 ib | ADD r/m32,imm8 | 2/7 | Add sign-extended immediate byte to r/m dword |
| 00 /r | ADD r/m8,r8 | 2/7 | Add byte register to r/m byte |
| 01 /r | ADD r/m16,r16 | 2/7 | Add word register to r/m word |
| 01 /r | ADD r/m32,r32 | 2/7 | Add dword register to r/m dword |
| 02 /r | ADD r8,r/m8 | 2/6 | Add r/m byte to byte register |
| 03 /r | ADD r16,r/m16 | 2/6 | Add r/m word to word register |
| 03 /r | ADD r32,r/m32 | 2/6 | Add r/m dword to dword register |

Operation

DEST ← DEST + SRC;

Description

ADD performs an integer addition of the two operands (DEST and SRC). The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte is added to a word or doubleword operand, the immediate value is sign-extended to the size of the word or doubleword operand.

Flags Affected

OF, SF, ZF, AF, CF, and PF as described in Appendix C

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

**AND —— Logical AND**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 24 ib | AND AL,imm8 | 2 | AND immediate byte to AL |
| 25 iw | AND AX,imm16 | 2 | AND immediate word to AX |
| 25 id | AND EAX,imm32 | 2 | AND immediate dword to EAX |
| 80 /4 ib | AND r/m8,imm8 | 2/7 | AND immediate byte to r/m byte |
| 81 /4 iw | AND r/m16,imm16 | 2/7 | AND immediate word to r/m word |
| 81 /4 id | AND r/m32,imm32 | 2/7 | AND immediate dword to r/m dword |
| 83 /4 ib | AND r/m16,imm8 | 2/7 | AND sign-extended immediate byte with r/m word |
| 83 /4 ib | AND r/m32,imm8 | 2/7 | AND sign-extended immediate byte with r/m dword |
| 20 /r | AND r/m8,r8 | 2/7 | AND byte register to r/m byte |
| 21 /r | AND r/m16,r16 | 2/7 | AND word register to r/m word |
| 21 /r | AND r/m32,r32 | 2/7 | AND dword register to r/m dword |
| 22 /r | AND r8,r/m8 | 2/6 | AND r/m byte to byte register |
| 23 /r | AND r16,r/m16 | 2/6 | AND r/m word to word register |
| 23 /r | AND r32,r/m32 | 2/6 | AND r/m dword to dword register |

Operation

```
DEST ← DEST AND SRC;
CF ← 0;
OF ← 0;
```

Description

Each bit of the result of the AND instruction is a 1 if both corresponding bits of the operands are 1; otherwise, it becomes a 0.

Flags Affected

CF = 0, OF = 0; PF, SF, and ZF as described in Appendix C

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

## ARPL —— Adjust RPL Field of Selector

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 63 /r | ARPL r/m16,r16 | pm=20/21 | Adjust RPL of r/m16 to not less than RPL of r16 |

Operation

```
IF RPL bits(0,1) of DEST < RPL bits(0,1) of SRC
THEN
   ZF ← 1;
   RPL bits(0,1) of DEST ← RPL bits(0,1) of SRC;
ELSE
   ZF ← 0;
FI;
```

Description

The ARPL instruction has two operands. The first operand is a 16-bit memory variable or word register that contains the value of a selector. The second operand is a word register. If the RPL field ("requested privilege level"——bottom two bits) of the first operand is less than the RPL field of the second operand, the zero flag is set to 1 and the RPL field of the first operand is increased to match the second operand. Otherwise, the zero flag is set to 0 and no change is made to the first operand.

ARPL appears in operating system software, not in application programs. It is used to guarantee that a selector parameter to a subroutine does not request more privilege than the caller is allowed. The second operand of ARPL is normally a register that contains the CS selector value of the caller.

Flags Affected

ZF as described above

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 6; ARPL is not recognized in Real Address Mode

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**BOUND —— Check Array Index Against Bounds**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 62 /r | BOUND r16,m16&16 | 10 | Check if r16 is within bounds (passes test) |
| 62 /r | BOUND r32,m32&32 | 10 | Check if r32 is within bounds (passes test) |

Operation

```
IF (LeftSRC < [RightSRC] OR LeftSRC > [RightSRC + OperandSize/8])
   (* Under lower bound or over upper bound *)
THEN Interrupt 5;
FI;
```

Description

BOUND ensures that a signed array index is within the limits specified by a block of memory consisting of an upper and a lower bound. Each bound uses one word for an operand-size attribute of 16 bits and a doubleword for an operand-size attribute of 32 bits. The first operand (a register) must be greater than or equal to the first bound in memory (lower bound), and less than or equal to the second bound in memory (upper bound). If the register is not within bounds, an Interrupt 5 occurs; the return EIP points to the BOUND instruction.

The bounds limit data structure is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array.

Flags Affected

None

Protected Mode Exceptions

Interrupt 5 if the bounds test fails, as described above; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

The second operand must be a memory operand, not a register. If BOUND is executed with a ModRM byte representing a register as the second operand, #UD occurs.

Real Address Mode Exceptions

Interrupt 5 if the bounds test fails; Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; Interrupt 6 if the second operand is a register

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## BSF —— Bit Scan Forward

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F BC | BSF r16,r/m16 | 10+3n | Bit scan forward on r/m word |
| 0F BC | BSF r32,r/m32 | 10+3n | Bit scan forward on r/m dword |

### Operation

```
IF r/m = 0
THEN
   ZF ← 1;
   register ← UNDEFINED;
ELSE
   temp ← 0;
   ZF ← 0;
   WHILE BIT[r/m, temp = 0]
   DO
      temp ← temp + 1;
      register ← temp;
   OD;
FI;
```

### Description

BSF scans the bits in the second word or doubleword operand starting with bit 0. The ZF flag is cleared if the bits are all 0; otherwise, the ZF flag is set and the destination register is loaded with the bit index of the first set bit.

### Flags Affected

ZF as described above

### Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

### Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

### Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

**BSR —— Bit Scan Reverse**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F BD | BSR r16,r/m16 | 10+3n | Bit scan reverse on r/m word |
| 0F BD | BSR r32,r/m32 | 10+3n | Bit scan reverse on r/m dword |

Operation

```
IF r/m = 0
THEN
   ZF ← 1;
   register ← UNDEFINED;
ELSE
   temp ← OperandSize - 1;
   ZF ← 0;
   WHILE BIT[r/m, temp] = 0
   DO
      temp ← temp - 1;
      register ← temp;
   OD;
FI;
```

Description

BSR scans the bits in the second word or doubleword operand from the most significant bit to the least significant bit. The ZF flag is cleared if the bits are all 0; otherwise, ZF is set and the destination register is loaded with the bit index of the first set bit found when scanning in the reverse direction.

Flags Affected

ZF as described above

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

**BT —— Bit Test**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F A3 | BT r/m16,r16 | 3/12 | Save bit in carry flag |
| 0F A3 | BT r/m32,r32 | 3/12 | Save bit in carry flag |
| 0F BA /4 ib | BT r/m16,imm8 | 3/6 | Save bit in carry flag |
| 0F BA /4 ib | BT r/m32,imm8 | 3/6 | Save bit in carry flag |

Operation

```
CF ← BIT[LeftSRC, RightSRC];
```

Description

BT saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag.

Flags Affected

CF as described above

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value is used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword. Immediate bit offsets larger than 31 are supported by using the immediate bit offset field in combination with the displacement field of the memory operand. The low-order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the high-order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode.

When accessing a bit in memory, the 80386 may access four bytes starting from the memory address given by:

```
Effective Address + (4 * (BitOffset DIV 32))
```

for a 32-bit operand size, or two bytes starting from the memory address given by:

```
Effective Address + (2 * (BitOffset DIV 16))
```

for a 16-bit operand size. It may do so even when only a single byte needs to be accessed in order to reach the given bit. You must therefore avoid referencing areas of memory close to address space holes. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

**BTC —— Bit Test and Complement**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F BB | BTC r/m16,r16 | 6/13 | Save bit in carry flag and complement |
| 0F BB | BTC r/m32,r32 | 6/13 | Save bit in carry flag and complement |
| 0F BA /7 ib | BTC r/m16,imm8 | 6/8 | Save bit in carry flag and complement |
| 0F BA /7 ib | BTC r/m32,imm8 | 6/8 | Save bit in carry flag and complement |

Operation

```
CF ← BIT[LeftSRC, RightSRC];
BIT[LeftSRC, RightSRC] ← NOT BIT[LeftSRC, RightSRC];
```

Description

BTC saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then complements the bit.

Flags Affected

CF as described above

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value is used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword. Immediate bit offsets larger than 31 are supported by using the immediate bit offset field in

combination with the displacement field of the memory operand. The low-order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the high-order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode.

When accessing a bit in memory, the 80386 may access four bytes starting from the memory address given by:

```
Effective Address + (4 * (BitOffset DIV 32))
```

for a 32-bit operand size, or two bytes starting from the memory address given by:

```
Effective Address + (2 * (BitOffset DIV 16))
```

for a 16-bit operand size. It may do so even when only a single byte needs to be accessed in order to reach the given bit. You must therefore avoid referencing areas of memory close to address space holes. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

**BTR —— Bit Test and Reset**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F B3 | BTR r/m16,r16 | 6/13 | Save bit in carry flag and reset |
| 0F B3 | BTR r/m32,r32 | 6/13 | Save bit in carry flag and reset |
| 0F BA /6 ib | BTR r/m16,imm8 | 6/8 | Save bit in carry flag and reset |
| 0F BA /6 ib | BTR r/m32,imm8 | 6/8 | Save bit in carry flag and reset |

Operation

```
CF ← BIT[LeftSRC, RightSRC];
BIT[LeftSRC, RightSRC] ← 0;
```

Description

BTR saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then stores 0 in the bit.

Flags Affected

CF as described above

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value is used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword. Immediate bit offsets larger than 31 (or 15) are supported by using the immediate bit offset field in combination with the displacement field of the memory operand. The

low-order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the high-order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode.

When accessing a bit in memory, the 80386 may access four bytes starting from the memory address given by:

```
Effective Address + 4 * (BitOffset DIV 32)
```

for a 32-bit operand size, or two bytes starting from the memory address given by:

```
Effective Address + 2 * (BitOffset DIV 16)
```

for a 16-bit operand size. It may do so even when only a single byte needs to be accessed in order to reach the given bit. You must therefore avoid referencing areas of memory close to address space holes. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

**BTS —— Bit Test and Set**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F AB | BTS r/m16,r16 | 6/13 | Save bit in carry flag and set |
| 0F AB | BTS r/m32,r32 | 6/13 | Save bit in carry flag and set |
| 0F BA /5 ib | BTS r/m16,imm8 | 6/8 | Save bit in carry flag and set |
| 0F BA /5 ib | BTS r/m32,imm8 | 6/8 | Save bit in carry flag and set |

Operation

```
CF ← BIT[LeftSRC, RightSRC];
BIT[LeftSRC, RightSRC] ← 1;
```

Description

BTS saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then stores 1 in the bit.

Flags Affected

CF as described above

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value is used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword. Immediate bit offsets larger than 31 are supported by using the immediate bit offset field in combination with the displacement field of the memory operand. The

low-order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the high order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode.

When accessing a bit in memory, the processor may access four bytes starting from the memory address given by:

```
Effective Address + (4 * (BitOffset DIV 32))
```

for a 32-bit operand size, or two bytes starting from the memory address given by:

```
Effective Address + (2 * (BitOffset DIV 16))
```

for a 16-bit operand size. It may do this even when only a single byte needs to be accessed in order to get at the given bit. Thus the programmer must be careful to avoid referencing areas of memory close to address space holes. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

**CALL — Call Procedure**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| E8 cw | CALL rel16 | 7+m | Call near, displacement relative to next instruction |
| FF /2 | CALL r/m16 | 7+m/10+m | Call near, register indirect/memory indirect |
| 9A cd | CALL ptr16:16 | 17+m,pm=34+m | Call intersegment, to full pointer given |
| 9A cd | CALL ptr16:16 | pm=52+m | Call gate, same privilege |
| 9A cd | CALL ptr16:16 | pm=86+m | Call gate, more privilege, no parameters |
| 9A cd | CALL ptr16:16 | pm=94+4x+m | Call gate, more privilege, x parameters |
| 9A cd | CALL ptr16:16 | ts | Call to task |
| FF /3 | CALL m16:16 | 22+m,pm=38+m | Call intersegment, address at r/m dword |
| FF /3 | CALL m16:16 | pm=56+m | Call gate, same privilege |
| FF /3 | CALL m16:16 | pm=90+m | Call gate, more privilege, no parameters |
| FF /3 | CALL m16:16 | pm=98+4x+m | Call gate, more privilege, x parameters |
| FF /3 | CALL m16:16 | 5 + ts | Call to task |
| E8 cd | CALL rel32 | 7+m | Call near, displacement relative to next instruction |
| FF /2 | CALL r/m32 | 7+m/10+m | Call near, indirect |
| 9A cp | CALL ptr16:32 | 17+m,pm=34+m | Call intersegment, to pointer given |
| 9A cp | CALL ptr16:32 | pm=52+m | Call gate, same privilege |
| 9A cp | CALL ptr16:32 | pm=86+m | Call gate, more privilege, no parameters |
| 9A cp | CALL ptr32:32 | pm=94+4x+m | Call gate, more privilege, x parameters |
| 9A cp | CALL ptr16:32 | ts | Call to task |
| FF /3 | CALL m16:32 | 22+m,pm=38+m | Call intersegment, address at r/m dword |
| FF /3 | CALL m16:32 | pm=56+m | Call gate, same privilege |
| FF /3 | CALL m16:32 | pm=90+m | Call gate, more privilege, no parameters |
| FF /3 | CALL m16:32 | pm=98+4x+m | Call gate, more privilege, x parameters |
| FF /3 | CALL m16:32 | 5 + ts | Call to task |

---

NOTE:
Values of ts are given by the following table:

| Old Task | New Task: 386 TSS VM = 0 | | New Task: 386 TSS VM = 1 | | New Task: 286 TSS | |
|---|---|---|---|---|---|---|
| | Via Task Gate? | | | | | |
| | N | Y | N | Y | N | Y |
| 386 TSS VM=0 | 300 | 309 | 217 | 226 | 273 | 282 |
| 286 TSS | 298 | 307 | 217 | 226 | 273 | 282 |

---

Operation

```
IF rel16 or rel32 type of call
THEN (* near relative call *)
   IF OperandSize = 16
   THEN
      Push(IP);
      EIP ← (EIP + rel16) AND 0000FFFFH;
   ELSE (* OperandSize = 32 *)
      Push(EIP);
      EIP ← EIP + rel32;
   FI;
FI;
```

```
IF r/m16 or r/m32 type of call
THEN (* near absolute call *)
   IF OperandSize = 16
   THEN
      Push(IP);
      EIP ← [r/m16] AND 0000FFFFH;
   ELSE (* OperandSize = 32 *)
      Push(EIP);
      EIP ← [r/m32];
   FI;
FI;

IF (PE = 0 OR (PE = 1 AND VM = 1))
(* real mode or virtual 8086 mode *)
   AND instruction = far CALL
   (* i.e., operand type is m16:16, m16:32, ptr16:16, ptr16:32 *)
THEN
   IF OperandSize = 16
   THEN
      Push(CS);
      Push(IP); (* address of next instruction; 16 bits *)
   ELSE
      Push(CS); (* padded with 16 high-order bits *)
      Push(EIP); (* address of next instruction; 32 bits *)
   FI;
   IF operand type is m16:16 or m16:32
   THEN (* indirect far call *)
      IF OperandSize = 16
      THEN
         CS:IP ← [m16:16];
         EIP ← EIP AND 0000FFFFH; (* clear upper 16 bits *)
      ELSE (* OperandSize = 32 *)
         CS:EIP ← [m16:32];
      FI;
   FI;
   IF operand type is ptr16:16 or ptr16:32
   THEN (* direct far call *)
      IF OperandSize = 16
      THEN
         CS:IP ← ptr16:16;
         EIP ← EIP AND 0000FFFFH; (* clear upper 16 bits *)
      ELSE (* OperandSize = 32 *)
         CS:EIP ← ptr16:32;
      FI;
   FI;
FI;

IF (PE = 1 AND VM = 0) (* Protected mode, not V86 mode *)
   AND instruction = far CALL
THEN
   If indirect, then check access of EA doubleword;
      #GP(0) if limit violation;
   New CS selector must not be null else #GP(0);
   Check that new CS selector index is within its
      descriptor table limits; else #GP(new CS selector);
   Examine AR byte of selected descriptor for various legal values;
      depending on value:
```

```
      go to CONFORMING-CODE-SEGMENT;
      go to NONCONFORMING-CODE-SEGMENT;
      go to CALL-GATE;
      go to TASK-GATE;
      go to TASK-STATE-SEGMENT;
   ELSE #GP(code segment selector);
FI;

CONFORMING-CODE-SEGMENT:
   DPL must be ≤ CPL ELSE #GP(code segment selector);
   Segment must be present ELSE #NP(code segment selector);
   Stack must be big enough for return address ELSE #SS(0);
   Instruction pointer must be in code segment limit ELSE #GP(0);
   Load code segment descriptor into CS register;
   Load CS with new code segment selector;
   Load EIP with zero-extend(new offset);
   IF OperandSize=16 THEN EIP ← EIP AND 0000FFFFH; FI;

NONCONFORMING-CODE-SEGMENT:
   RPL must be ≤ CPL ELSE #GP(code segment selector)
   DPL must be = CPL ELSE #GP(code segment selector)
   Segment must be present ELSE #NP(code segment selector)
   Stack must be big enough for return address ELSE #SS(0)
   Instruction pointer must be in code segment limit ELSE #GP(0)
   Load code segment descriptor into CS register
   Load CS with new code segment selector
   Set RPL of CS to CPL
   Load EIP with zero-extend(new offset);
   IF OperandSize=16 THEN EIP ← EIP AND 0000FFFFH; FI;

CALL-GATE:
   Call gate DPL must be ≥ CPL ELSE #GP(call gate selector)
   Call gate DPL must be ≥ RPL ELSE #GP(call gate selector)
   Call gate must be present ELSE #NP(call gate selector)
   Examine code segment selector in call gate descriptor:
      Selector must not be null ELSE #GP(0)
      Selector must be within its descriptor table
         limits ELSE #GP(code segment selector)
   AR byte of selected descriptor must indicate code
      segment ELSE #GP(code segment selector)
   DPL of selected descriptor must be ≤ CPL ELSE
      #GP(code segment selector)
   IF non-conforming code segment AND DPL < CPL
   THEN go to MORE-PRIVILEGE
   ELSE go to SAME-PRIVILEGE
   FI;

MORE-PRIVILEGE:
   Get new SS selector for new privilege level from TSS
      Check selector and descriptor for new SS:
         Selector must not be null ELSE #TS(0)
         Selector index must be within its descriptor
            table limits ELSE #TS(SS selector)
         Selector's RPL must equal DPL of code segment
            ELSE #TS(SS selector)
         Stack segment DPL must equal DPL of code
            segment ELSE #TS(SS selector)
```

```
         Descriptor must indicate writable data segment
            ELSE #TS(SS selector)
         Segment present ELSE #SS(SS selector)
      IF OperandSize=32
      THEN
         New stack must have room for parameters plus 16 bytes
            ELSE #SS(0)
         EIP must be in code segment limit ELSE #GP(0)
         Load new SS:eSP value from TSS
         Load new CS:EIP value from gate
      ELSE
         New stack must have room for parameters plus 8 bytes ELSE #SS(0)
         IP must be in code segment limit ELSE #GP(0)
         Load new SS:eSP value from TSS
         Load new CS:IP value from gate
      FI;
      Load CS descriptor
      Load SS descriptor
      Push long pointer of old stack onto new stack
      Get word count from call gate, mask to 5 bits
      Copy parameters from old stack onto new stack
      Push return address onto new stack
      Set CPL to stack segment DPL
      Set RPL of CS to CPL

SAME-PRIVILEGE:
   IF OperandSize=32
   THEN
      Stack must have room for 6-byte return address (padded to 8 bytes)
         ELSE #SS(0)
      EIP must be within code segment limit ELSE #GP(0)
      Load CS:EIP from gate
   ELSE
      Stack must have room for 4-byte return address ELSE #SS(0)
      IP must be within code segment limit ELSE #GP(0)
      Load CS:IP from gate
   FI;
   Push return address onto stack
   Load code segment descriptor into CS register
   Set RPL of CS to CPL

TASK-GATE:
   Task gate DPL must be ≥ CPL ELSE #TS(gate selector)
   Task gate DPL must be ≥ RPL ELSE #TS(gate selector)
   Task Gate must be present ELSE #NP(gate selector)
   Examine selector to TSS, given in Task Gate descriptor:
      Must specify global in the local/global bit ELSE #TS(TSS selector)
      Index must be within GDT limits ELSE #TS(TSS selector)
      TSS descriptor AR byte must specify nonbusy TSS
         ELSE #TS(TSS selector)
      Task State Segment must be present ELSE #NP(TSS selector)
   SWITCH-TASKS (with nesting) to TSS
   IP must be in code segment limit ELSE #TS(0)
```

```
TASK-STATE-SEGMENT:
   TSS DPL must be ≥ CPL else #TS(TSS selector)
   TSS DPL must be ≥ RPL ELSE #TS(TSS selector)
   TSS descriptor AR byte must specify available TSS
      ELSE #TS(TSS selector)
   Task State Segment must be present ELSE #NP(TSS selector)
   SWITCH-TASKS (with nesting) to TSS
   IP must be in code segment limit ELSE #TS(0)
```

Description

The CALL instruction causes the procedure named in the operand to be executed. When the procedure is complete (a return instruction is executed within the procedure), execution continues at the instruction that follows the CALL instruction.

The action of the different forms of the instruction are described below.

Near calls are those with destinations of type r/m16, r/m32, rel16, rel32; changing or saving the segment register value is not necessary. The CALL rel16 and CALL rel32 forms add a signed offset to the address of the instruction following CALL to determine the destination. The rel16 form is used when the instruction's operand-size attribute is 16 bits; rel32 is used when the operand-size attribute is 32 bits. The result is stored in the 32-bit EIP register. With rel16, the upper 16 bits of EIP are cleared, resulting in an offset whose value does not exceed 16 bits. CALL r/m16 and CALL r/m32 specify a register or memory location from which the absolute segment offset is fetched. The offset fetched from r/m is 32 bits for an operand-size attribute of 32 (r/m32), or 16 bits for an operand-size of 16 (r/m16). The offset of the instruction following CALL is pushed onto the stack. It will be popped by a near RET instruction within the procedure. The CS register is not changed by this form of CALL.

The far calls, CALL ptr16:16 and CALL ptr16:32, use a four-byte or six-byte operand as a long pointer to the procedure called. The CALL m16:16 and m16:32 forms fetch the long pointer from the memory location specified (indirection). In Real Address Mode or Virtual 8086 Mode, the long pointer provides 16 bits for the CS register and 16 or 32 bits for the EIP register (depending on the operand-size attribute). These forms of the instruction push both CS and IP or EIP as a return address.

In Protected Mode, both long pointer forms consult the AR byte in the descriptor indexed by the selector part of the long pointer. Depending on the value of the AR byte, the call will perform one of the following types of control transfers:

- A far call to the same protection level
- An inter-protection level far call
- A task switch

For more information on Protected Mode control transfers, refer to Chapter 6 and Chapter 7.

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur

Protected Mode Exceptions

For far calls: #GP, #NP, #SS, and #TS, as indicated in the list above

For near direct calls: #GP(0) if procedure location is beyond the code segment limits; #SS(0) if pushing the return address exceeds the bounds of the stack segment; #PF (fault-code) for a page fault

For a near indirect call: #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #GP(0) if the indirect offset obtained is beyond the code segment limits; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

Notes

Any far call from a 32-bit code segment to 16-bit code segments should be made from the first 64K bytes of the 32-bit code segment, since the operand-size attribute of the instruction is set to 16, thus allowing only a 16-bit return address offset to be saved.

## CBW/CWDE —— Convert Byte to Word/Convert Word to Doubleword

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 98 | CBW | 3 | AX ← sign-extend of AL |
| 98 | CWDE | 3 | EAX ← sign-extend of AX |

### Operation

```
IF OperandSize = 16 (* instruction = CBW *)
THEN AX ← SignExtend(AL);
ELSE (* OperandSize = 32, instruction = CWDE *)
   EAX ← SignExtend(AX);
FI;
```

### Description

CBW converts the signed byte in AL to a signed word in AX by extending the most significant bit of AL (the sign bit) into all of the bits of AH. CWDE converts the signed word in AX to a doubleword in EAX by extending the most significant bit of AX into the two most significant bytes of EAX. Note that CWDE is different from CWD. CWD uses DX:AX rather than EAX as a destination.

### Flags Affected

None

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

## CLC — Clear Carry Flag

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F8 | CLC | 2 | Clear carry flag |

### Operation

```
CF ← 0;
```

### Description

CLC sets the carry flag to zero. It does not affect other flags or registers.

### Flags Affected

CF = 0

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

## CLD —— Clear Direction Flag

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| FC | CLD | 2 | Clear direction flag; SI and DI will increment during string instructions |

### Operation

```
DF ← 0;
```

### Description

CLD clears the direction flag. No other flags or registers are affected. After CLD is executed, string operations will increment the index registers (SI and/or DI) that they use.

### Flags Affected

DF = 0

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

## CLI —— Clear Interrupt Flag

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| FA | CLI | 3 | Clear interrupt flag; interrupts disabled |

### Operation

```
IF ← 0;
```

### Description

CLI clears the interrupt flag if the current privilege level is at least as privileged as IOPL. No other flags are affected. External interrupts are not recognized at the end of the CLI instruction or from that point on until the interrupt flag is set.

### Flags Affected

IF = 0

### Protected Mode Exceptions

#GP(0) if the current privilege level is greater (has less privilege) than the IOPL in the flags register. IOPL specifies the least privileged level at which I/O can be performed.

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

#GP(0) as for Protected Mode

## CLTS —— Clear Task-Switched Flag in CR0

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| OF 06 | CLTS | 5 | Clear task-switched flag |

### Operation

```
TS Flag in CR0 ← 0;
```

### Description

CLTS clears the task-switched (TS) flag in register CR0. This flag is set by the 80386 every time a task switch occurs. The TS flag is used to manage processor extensions as follows:

- Every execution of an ESC instruction is trapped if the TS flag is set.
- Execution of a WAIT instruction is trapped if the MP flag and the TS flag are both set.

Thus, if a task switch was made after an ESC instruction was begun, the processor extension's context may need to be saved before a new ESC instruction can be issued. The fault handler saves the context and resets the TS flag.

CLTS appears in operating system software, not in application programs. It is a privileged instruction that can only be executed at privilege level 0.

### Flags Affected

TS = 0 (TS is in CR0, not the flag register)

### Protected Mode Exceptions

#GP(0) if CLTS is executed with a current privilege level other than 0

### Real Address Mode Exceptions

None (valid in Real Address Mode to allow initialization for Protected Mode)

### Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode

## CMC — Complement Carry Flag

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F5 | CMC | 2 | Complement carry flag |

### Operation

```
CF ← NOT CF;
```

### Description

CMC reverses the setting of the carry flag. No other flags are affected.

### Flags Affected

CF as described above

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

**CMP —— Compare Two Operands**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 3C ib | CMP AL,imm8 | 2 | Compare immediate byte to AL |
| 3D iw | CMP AX,imm16 | 2 | Compare immediate word to AX |
| 3D id | CMP EAX,imm32 | 2 | Compare immediate dword to EAX |
| 80 /7 ib | CMP r/m8,imm8 | 2/5 | Compare immediate byte to r/m byte |
| 81 /7 iw | CMP r/m16,imm16 | 2/5 | Compare immediate word to r/m word |
| 81 /7 id | CMP r/m32,imm32 | 2/5 | Compare immediate dword to r/m dword |
| 83 /7 ib | CMP r/m16,imm8 | 2/5 | Compare sign extended immediate byte to r/m word |
| 83 /7 ib | CMP r/m32,imm8 | 2/5 | Compare sign extended immediate byte to r/m dword |
| 38 /r | CMP r/m8,r8 | 2/5 | Compare byte register to r/m byte |
| 39 /r | CMP r/m16,r16 | 2/5 | Compare word register to r/m word |
| 39 /r | CMP r/m32,r32 | 2/5 | Compare dword register to r/m dword |
| 3A /r | CMP r8,r/m8 | 2/6 | Compare r/m byte to byte register |
| 3B /r | CMP r16,r/m16 | 2/6 | Compare r/m word to word register |
| 3B /r | CMP r32,r/m32 | 2/6 | Compare r/m dword to dword register |

Operation

```
LeftSRC - SignExtend(RightSRC);
(* CMP does not store a result; its purpose is to set the flags *)
```

Description

CMP subtracts the second operand from the first but, unlike the SUB instruction, does not store the result; only the flags are changed. CMP is typically used in conjunction with conditional jumps and the SETcc instruction. (Refer to Appendix D for the list of signed and unsigned flag tests provided.) If an operand greater than one byte is compared to an immediate byte, the byte value is first sign-extended.

Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix C

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

## CMPS/CMPSB/CMPSW/CMPSD —— Compare String Operands

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| A6 | CMPS m8,m8 | 10 | Compare bytes ES:[(E)DI] (second operand) with [(E)SI] (first operand) |
| A7 | CMPS m16,m16 | 10 | Compare words ES:[(E)DI] (second operand) with [(E)SI] (first operand) |
| A7 | CMPS m32,m32 | 10 | Compare dwords ES:[(E)DI] (second operand) with [(E)SI] (first operand) |
| A6 | CMPSB | 10 | Compare bytes ES:[(E)DI] with DS:[SI] |
| A7 | CMPSW | 10 | Compare words ES:[(E)DI] DS:[SI] |
| A7 | CMPSD | 10 | Compare dwords ES:[(E)DI] with DS:[SI] |

Operation

```
IF (instruction = CMPSD) OR
   (instruction has operands of type DWORD)
THEN OperandSize ← 32;
ELSE OperandSize ← 16;
FI;
IF AddressSize = 16
THEN
   use SI for source-index and DI for destination-index
ELSE (* AddressSize = 32 *)
   use ESI for source-index and EDI for destination-index;
FI;
IF byte type of instruction
THEN
   [source-index] - [destination-index]; (* byte comparison *)
   IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← -1; FI;
ELSE
   IF OperandSize = 16
   THEN
      [source-index] - [destination-index]; (* word comparison *)
      IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← -2; FI;
   ELSE (* OperandSize = 32 *)
      [source-index] - [destination-index]; (* dword comparison *)
      IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← -4; FI;
   FI;
FI;
source-index = source-index + IncDec;
destination-index = destination-index + IncDec;
```

Description

CMPS compares the byte, word, or doubleword pointed to by the source-index register with the byte, word, or doubleword pointed to by the destination-index register.

If the address-size attribute of this instruction is 16 bits, SI and DI will be used for source- and destination-index registers; otherwise ESI and EDI will be used. Load the correct index values into SI and DI (or ESI and EDI) before executing CMPS.

The comparison is done by subtracting the operand indexed by the destination-index register from the operand indexed by the source-index register.

Note that the direction of subtraction for CMPS is [SI] - [DI] or [ESI] - [EDI]. The left operand (SI or ESI) is the source and the right operand (DI or EDI) is the destination. This is the reverse of the usual Intel convention in which the left operand is the destination and the right operand is the source.

The result of the subtraction is not stored; only the flags reflect the change. The types of the operands determine whether bytes, words, or doublewords are compared. For the first operand (SI or ESI), the DS register is used, unless a segment override byte is present. The second operand (DI or EDI) must be addressable from the ES register; no segment override is possible.

After the comparison is made, both the source-index register and destination-index register are automatically advanced. If the direction flag is 0 (CLD was executed), the registers increment; if the direction flag is 1 (STD was executed), the registers decrement. The registers increment or decrement by 1 if a byte is compared, by 2 if a word is compared, or by 4 if a doubleword is compared.

CMPSB, CMPSW and CMPSD are synonyms for the byte, word, and doubleword CMPS instructions, respectively.

CMPS can be preceded by the REPE or REPNE prefix for block comparison of CX or ECX bytes, words, or doublewords. Refer to the description of the REP instruction for more information on this operation.

Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix C

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault

## CWD/CDQ —— Convert Word to Doubleword/Convert Doubleword to Quadword

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 99 | CWD | 2 | DX:AX ← sign-extend of AX |
| 99 | CDQ | 2 | EDX:EAX ← sign-extend of EAX |

### Operation

```
IF OperandSize = 16 (* CWD instruction *)
THEN
   IF AX < 0 THEN DX ← 0FFFFH; ELSE DX ← 0; FI;
ELSE (* OperandSize = 32, CDQ instruction *)
   IF EAX < 0 THEN EDX ← 0FFFFFFFFH; ELSE EDX ← 0; FI;
FI;
```

### Description

CWD converts the signed word in AX to a signed doubleword in DX:AX by extending the most significant bit of AX into all the bits of DX. CDQ converts the signed doubleword in EAX to a signed 64-bit integer in the register pair EDX:EAX by extending the most significant bit of EAX (the sign bit) into all the bits of EDX. Note that CWD is different from CWDE. CWDE uses EAX as a destination, instead of DX:AX.

### Flags Affected

None

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

## DAA — Decimal Adjust AL after Addition

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 27 | DAA | 4 | Decimal adjust AL after addition |

### Operation

```
IF ((AL AND 0FH) > 9) OR (AF = 1)
THEN
   AL ← AL + 6;
   AF ← 1;
ELSE
   AF ← 0;
FI;
IF (AL > 9FH) OR (CF = 1)
THEN
   AL ← AL + 60H;
   CF ← 1;
ELSE CF ← 0;
FI;
```

### Description

Execute DAA only after executing an ADD instruction that leaves a two-BCD-digit byte result in the AL register. The ADD operands should consist of two packed BCD digits. The DAA instruction adjusts AL to contain the correct two-digit packed decimal result.

### Flags Affected

AF and CF as described above; SF, ZF, PF, and CF as described in Appendix C.

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

**DAS —— Decimal Adjust AL after Subtraction**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 2F | DAS | 4 | Decimal adjust AL after subtraction |

Operation

```
IF (AL AND 0FH) > 9 OR AF = 1
THEN
   AL ← AL - 6;
   AF ← 1;
ELSE
   AF ← 0;
FI;
IF (AL > 9FH) OR (CF = 1)
THEN
   AL ← AL - 60H;
   CF ← 1;
ELSE CF ← 0;
FI;
```

Description

Execute DAS only after a subtraction instruction that leaves a two-BCD-digit byte result in the AL register. The operands should consist of two packed BCD digits. DAS adjusts AL to contain the correct packed two-digit decimal result.

Flags Affected

AF and CF as described above; SF, ZF, and PF as described in Appendix C.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

**DEC —— Decrement by 1**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| FE /1 | DEC r/m8 | 2/6 | Decrement r/m byte by 1 |
| FF /1 | DEC r/m16 | 2/6 | Decrement r/m word by 1 |
| | DEC r/m32 | 2/6 | Decrement r/m dword by 1 |
| 48+rw | DEC r16 | 2 | Decrement word register by 1 |
| 48+rw | DEC r32 | 2 | Decrement dword register by 1 |

Operation

DEST ← DEST - 1;

Description

DEC subtracts 1 from the operand. DEC does not change the carry flag. To affect the carry flag, use the SUB instruction with an immediate operand of 1.

Flags Affected

OF, SF, ZF, AF, and PF as described in Appendix C.

Protected Mode Exceptions

#GP(0) if the result is a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**DIV —— Unsigned Divide**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F6 /6 | DIV AL,r/m8 | 14/17 | Unsigned divide AX by r/m byte (AL=Quo, AH=Rem) |
| F7 /6 | DIV AX,r/m16 | 22/25 | Unsigned divide DX:AX by r/m word (AX=Quo, DX=Rem) |
| F7 /6 | DIV EAX,r/m32 | 38/41 | Unsigned divide EDX:EAX by r/m dword (EAX=Quo, EDX=Rem) |

Operation

```
temp ← dividend / divisor;
IF temp does not fit in quotient
THEN Interrupt 0;
ELSE
   quotient ← temp;
   remainder ← dividend MOD (r/m);
FI;
```

---

Note:
  Divisions are unsigned. The divisor is given by the r/m operand.
  The dividend, quotient, and remainder use implicit registers. Refer to
  the table under "Description."

---

Description

DIV performs an unsigned division. The dividend is implicit; only the divisor is given as an operand. The remainder is always less than the divisor. The type of the divisor determines which registers to use:

| Size | Dividend | Divisor | Quotient | Remainder |
|---|---|---|---|---|
| byte | AX | r/m8 | AL | AH |
| word | DX:AX | r/m16 | AX | DX |
| dword | EDX:EAX | r/m32 | EAX | EDX |

Flags Affected

OF, SF, ZF, AR, PF, CF are undefined.

Protected Mode Exceptions

Interrupt 0 if the quotient is too large to fit in the designated register (AL, AX, or EAX), or if the divisor is 0; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 0 if the quotient is too big to fit in the designated register (AL, AX, or EAX), or if the divisor is 0; Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

**ENTER —— Make Stack Frame for Procedure Parameters**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| C8 iw 00 | ENTER imm16,0 | 10 | Make procedure stack frame |
| C8 iw 01 | ENTER imm16,1 | 12 | Make stack frame for procedure parameters |
| C8 iw ib | ENTER imm16,imm8 | 15+4(n-1) | Make stack frame for procedure parameters |

Operation

```
level ← level MOD 32
IF OperandSize = 16 THEN Push(BP) ELSE Push (EBP) FI;
   (* Save stack pointer *)
frame-ptr ← eSP
IF level > 0
THEN (* level is rightmost parameter *)
   FOR i ← 1 TO level - 1
   DO
      IF OperandSize = 16
      THEN
         BP ← BP - 2;
         Push[BP]
      ELSE (* OperandSize = 32 *)
         EBP ← EBP - 4;
         Push[EBP];
      FI;
   OD;
   Push(frame-ptr)
FI;
IF OperandSize = 16 THEN BP ← frame-ptr ELSE EBP ← frame-ptr; FI;
IF StackAddrSize = 16
THEN SP ← SP - First operand;
ELSE ESP ← ESP - ZeroExtend(First operand);
FI;
```

Description

ENTER creates the stack frame required by most block-structured high-level languages. The first operand specifies the number of bytes of dynamic storage allocated on the stack for the routine being entered. The second operand gives the lexical nesting level (0 to 31) of the routine within the high-level language source code. It determines the number of stack frame pointers copied into the new stack frame from the preceding frame. BP (or EBP, if the operand-size attribute is 32 bits) is the current stack frame pointer.

If the operand-size attribute is 16 bits, the processor uses BP as the frame pointer and SP as the stack pointer. If the operand-size attribute is 32 bits, the processor uses EBP for the frame pointer and ESP for the stack pointer.

If the second operand is 0, ENTER pushes the frame pointer (BP or EBP) onto the stack; ENTER then subtracts the first operand from the stack pointer and sets the frame pointer to the current stack-pointer value.

For example, a procedure with 12 bytes of local variables would have an ENTER 12,0 instruction at its entry point and a LEAVE instruction before every RET. The 12 local bytes would be addressed as negative offsets from the frame pointer.

Flags Affected

None

Protected Mode Exceptions

#SS(0) if SP or ESP would exceed the stack limit at any point during instruction execution; #PF(fault-code) for a page fault

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

**HLT —— Halt**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F4 | HLT | 5 | Halt |

Operation

Enter Halt state;

Description

HALT stops instruction execution and places the 80386 in a HALT state. An enabled interrupt, NMI, or a reset will resume execution. If an interrupt (including NMI) is used to resume execution after HLT, the saved CS:IP (or CS:EIP) value points to the instruction following HLT.

Flags Affected

None

Protected Mode Exceptions

HLT is a privileged instruction; #GP(0) if the current privilege level is not 0

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

#GP(0); HLT is a privileged instruction

**IDIV —— Signed Divide**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F6 /7 | IDIV r/m8 | 19 | Signed divide AX by r/m byte (AL=Quo, AH=Rem) |
| F7 /7 | IDIV AX,r/m16 | 27 | Signed divide DX:AX by EA word (AX=Quo, DX=Rem) |
| F7 /7 | IDIV EAX,r/m32 | 43 | Signed divide EDX:EAX by DWORD byte (EAX=Quo, EDX=Rem) |

Operation

```
temp ← dividend / divisor;
IF temp does not fit in quotient
THEN Interrupt 0;
ELSE
   quotient ← temp;
   remainder ← dividend MOD (r/m);
FI;
```

---

Notes:
Divisions are signed. The divisor is given by the r/m operand. The dividend, quotient, and remainder use implicit registers. Refer to the table under "Description."

---

Description

IDIV performs a signed division. The dividend, quotient, and remainder are implicitly allocated to fixed registers. Only the divisor is given as an explicit r/m operand. The type of the divisor determines which registers to use as follows:

| Size | Divisor | Quotient | Remainder | Dividend |
|---|---|---|---|---|
| byte | r/m8 | AL | AH | AX |
| word | r/m16 | AX | DX | DX:AX |
| dword | r/m32 | EAX | EDX | EDX:EAX |

If the resulting quotient is too large to fit in the destination, or if the division is 0, an Interrupt 0 is generated. Nonintegral quotients are truncated toward 0. The remainder has the same sign as the dividend and the absolute value of the remainder is always less than the absolute value of the divisor.

Flags Affected

OF, SF, ZF, AR, PF, CF are undefined.

Protected Mode Exceptions

Interrupt 0 if the quotient is too large to fit in the designated register (AL or AX), or if the divisor is 0; #GP (0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 0 if the quotient is too large to fit in the designated register (AL or AX), or if the divisor is 0; Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**IMUL —— Signed Multiply**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F6 /5 | IMUL r/m8 | 9-14/12-17 | AX← AL * r/m byte |
| F7 /5 | IMUL r/m16 | 9-22/12-25 | DX:AX ← AX * r/m word |
| F7 /5 | IMUL r/m32 | 9-38/12-41 | EDX:EAX ← EAX * r/m dword |
| 0F AF /r | IMUL r16,r/m16 | 9-22/12-25 | word register ← word register * r/m word |
| 0F AF /r | IMUL r32,r/m32 | 9-38/12-41 | dword register ← dword register * r/m dword |
| 6B /r ib | IMUL r16,r/m16,imm8 | 9-14/12-17 | word register ← r/m16 * sign-extended immediate byte |
| 6B /r ib | IMUL r32,r/m32,imm8 | 9-14/12-17 | dword register ← r/m32 * sign-extended immediate byte |
| 6B /r ib | IMUL r16,imm8 | 9-14/12-17 | word register ← word register * sign-extended immediate byte |
| 6B /r ib | IMUL r32,imm8 | 9-14/12-17 | dword register ← dword register * sign-extended immediate byte |
| 69 /r iw | IMUL r16,r/m16,imm16 | 9-22/12-25 | word register ← r/m16 * immediate word |
| 69 /r id | IMUL r32,r/m32,imm32 | 9-38/12-41 | dword register ← r/m32 * immediate dword |
| 69 /r iw | IMUL r16,imm16 | 9-22/12-25 | word register ← r/m16 * immediate word |
| 69 /r id | IMUL r32,imm32 | 9-38/12-41 | dword register ← r/m32 * immediate dword |

---

NOTES:
The 80386 uses an early-out multiply algorithm. The actual number of clocks depends on the position of the most significant bit in the optimizing multiplier, shown underlined above. The optimization occurs for positive and negative values. Because of the early-out algorithm, clock counts given are minimum to maximum. To calculate the actual clocks, use the following formula:

---

Actual clock = if $m \neq 0$ then $\max(\text{ceiling}(\log_2 |m|), 3) + 6$ clocks
Actual clock = if $m = 0$ then 9 clocks
(where m is the multiplier)

Add three clocks if the multiplier is a memory operand.

Operation

```
result ← multiplicand * multiplier;
```

Description

IMUL performs signed multiplication. Some forms of the instruction use implicit register operands. The operand combinations for all forms of the instruction are shown in the "Description" column above.

IMUL clears the overflow and carry flags under the following conditions:

| Instruction Form | Condition for Clearing CF and OF |
|---|---|
| r/m8 | AL = sign-extend of AL to 16 bits |
| r/m16 | AX = sign-extend of AX to 32 bits |
| r/m32 | EDX:EAX = sign-extend of EAX to 32 bits |
| r16,r/m16 | Result exactly fits within r16 |
| r/32,r/m32 | Result exactly fits within r32 |
| r16,r/m16,imm16 | Result exactly fits within r16 |
| r32,r/m32,imm32 | Result exactly fits within r32 |

Flags Affected

OF and CF as described above; SF, ZF, AF, and PF are undefined

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exeptions as in Real Address Mode; #PF(fault-code) for a page fault

Notes

When using the accumulator forms (IMUL r/m8, IMUL r/m16, or IMUL r/m32), the result of the multiplication is available even if the overflow flag is set because the result is two times the size of the multiplicand and multiplier. This is large enough to handle any possible result.

**IN —— Input from Port**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| E4 ib | IN AL,imm8 | 12,pm=6*/26** | Input byte from immediate port into AL |
| E5 ib | IN AX,imm8 | 12,pm=6*/26** | Input word from immediate port into AX |
| E5 ib | IN EAX,imm8 | 12,pm=6*/26** | Input dword from immediate port into EAX |
| EC | IN AL,DX | 13,pm=7*/27** | Input byte from port DX into AL |
| ED | IN AX,DX | 13,pm=7*/27** | Input word from port DX into AX |
| ED | IN EAX,DX | 13,pm=7*/27** | Input dword from port DX into EAX |

---

NOTES:
*If CPL ≤ IOPL
**If CPL > IOPL or if in virtual 8086 mode

---

Operation

```
IF (PE = 1) AND ((VM = 1) OR (CPL > IOPL))
THEN (* Virtual 8086 mode, or protected mode with CPL > IOPL *)
   IF NOT I-O-Permission (SRC, width(SRC))
   THEN #GP(0);
   FI;
FI;
DEST ← [SRC]; (* Reads from I/O address space *)
```

Description

IN transfers a data byte or data word from the port numbered by the second operand into the register (AL, AX, or EAX) specified by the first operand. Access any port from 0 to 65535 by placing the port number in the DX register and using an IN instruction with DX as the second parameter. These I/O instructions can be shortened by using an 8-bit port I/O in the instruction. The upper eight bits of the port address will be 0 when 8-bit port I/O is used.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if the current privilege level is larger (has less privilege) than IOPL and any of the corresponding I/O permission bits in TSS equals 1

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

#GP(0) fault if any of the corresponding I/O permission bits in TSS equals 1

## INC —— Increment by 1

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| FE /0 | INC r/m8 | | Increment r/m byte by 1 |
| FF /0 | INC r/m16 | | Increment r/m word by 1 |
| FF /6 | INC r/m32 | | Increment r/m dword by 1 |
| 40 + rw | INC r16 | | Increment word register by 1 |
| 40 + rd | INC r32 | | Increment dword register by 1 |

### Operation

DEST ← DEST + 1;

### Description

INC adds 1 to the operand. It does not change the carry flag. To affect the carry flag, use the ADD instruction with a second operand of 1.

### Flags Affected

OF, SF, ZF, AF, and PF as described in Appendix C

### Protected Mode Exceptions

#GP(0) if the operand is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

### Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

### Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## INS/INSB/INSW/INSD —— Input from Port to String

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 6C | INS r/m8,DX | 15,pm=9*/29** | Input byte from port DX into ES:(E)DI |
| 6D | INS r/m16,DX | 15,pm=9*/29** | Input word from port DX into ES:(E)DI |
| 6D | INS r/m32,DX | 15,pm=9*/29** | Input dword from port DX into ES:(E)DI |
| 6C | INSB | 15,pm=9*/29** | Input byte from port DX into ES:(E)DI |
| 6D | INSW | 15,pm=9*/29** | Input word from port DX into ES:(E)DI |
| 6D | INSD | 15,pm=9*/29** | Input dword from port DX into ES:(E)DI |

---

NOTES:
*If CPL ≤ IOPL
**If CPL > IOPL or if in virtual 8086 mode

---

Operation

```
IF AddressSize = 16
THEN use DI for dest-index;
ELSE (* AddressSize = 32 *)
   use EDI for dest-index;
FI;
IF (PE = 1) AND ((VM = 1) OR (CPL > IOPL))
THEN (* Virtual 8086 mode, or protected mode with CPL > IOPL *)
   IF NOT I-O-Permission (SRC, width(SRC))
   THEN #GP(0);
   FI;
FI;
IF byte type of instruction
THEN
   ES:[dest-index] ← [DX]; (* Reads byte at DX from I/O address space *)
   IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← -1; FI;
FI;
IF OperandSize = 16
THEN
   ES:[dest-index] ← [DX]; (* Reads word at DX from I/O address space *)
   IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← -2; FI;
FI;
IF OperandSize = 32
THEN
   ES:[dest-index] ← [DX]; (* Reads dword at DX from I/O address space *)
   IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← -4; FI;
FI;
dest-index ← dest-index + IncDec;
```

Description

INS transfers data from the input port numbered by the DX register to the memory byte or word at ES:dest-index. The memory operand must be addressable from ES; no segment override is possible. The destination register is DI if the address-size attribute of the instruction is 16 bits, or EDI if the address-size attribute is 32 bits.

INS does not allow the specification of the port number as an immediate value. The port must be addressed through the DX register value. Load the correct value into DX before executing the INS instruction.

The destination address is determined by the contents of the destination index register. Load the correct index into the destination index register before executing INS.

After the transfer is made, DI or EDI advances automatically. If the direction flag is 0 (CLD was executed), DI or EDI increments; if the direction flag is 1 (STD was executed), DI or EDI decrements. DI increments or decrements by 1 if a byte is input, by 2 if a word is input, or by 4 if a doubleword is input.

INSB, INSW and INSD are synonyms of the byte, word, and doubleword INS instructions. INS can be preceded by the REP prefix for block input of CX bytes or words. Refer to the REP instruction for details of this operation.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if CPL is numerically greater than IOPL and any of the corresponding I/O permission bits in TSS equals 1; #GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

#GP(0) fault if any of the corresponding I/O permission bits in TSS equals 1; #PF(fault-code) for a page fault

**INT/INTO —— Call to Interrupt Procedure**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| CC | INT 3 | 33 | Interrupt 3--trap to debugger |
| CC | INT 3 | pm=59 | Interrupt 3--Protected Mode, same privilege |
| CC | INT 3 | pm=99 | Interrupt 3--Protected Mode, more privilege |
| CC | INT 3 | pm=119 | Interrupt 3--from V86 mode to PL 0 |
| CC | INT 3 | ts | Interrupt 3--Protected Mode, via task gate |
| CD ib | INT imm8 | 37 | Interrupt numbered by byte |
| CD ib | INT imm8 | pm=59 | Interrupt--Protected Mode, same privilege |
| CD ib | INT imm8 | pm=99 | Interrupt--Protected Mode, more privilege |
| CD ib | INT imm8 | pm=119 | Interrupt--from V86 mode to PL 0 |
| CD ib | INT imm8 | ts | Interrupt--Protected Mode, via task gate |
| CE | INTO | Fail:3,pm=3; | |
| | | Pass:35 | Interrupt 4--if overflow flag is 1 |
| CE | INTO | pm=59 | Interrupt 4--Protected Mode, privilege |
| CE | INTO | pm=99 | Interrupt 4--Protected Mode, more privilege |
| CE | INTO | pm=119 | Interrupt 4--from V86 mode to PL 0 |
| CE | INTO | ts | Interrupt 4--Protected Mode, via task gate |

---

NOTE:
Approximate values of ts are given by the following table:

| | New Task | | |
|---|---|---|---|
| Old Task | 386 TSS VM = 0 | 386 TSS VM = 1 | 286 TSS |
| 386 TSS VM=0 | 309 | 226 | 282 |
| 386 TSS VM=1 | 314 | 231 | 287 |
| 286 TSS | 307 | 224 | 280 |

---

Operation

---

NOTE:
The following operational description applies not only to the above instructions but also to external interrupts and exceptions.

---

```
IF PE = 0
THEN GOTO REAL-ADDRESS-MODE;
ELSE GOTO PROTECTED-MODE;
FI;

REAL-ADDRESS-MODE:
   Push (FLAGS);
   IF ← 0; (* Clear interrupt flag *)
   TF ← 0; (* Clear trap flag *)
   Push(CS);
   Push(IP);
```

```
   (* No error codes are pushed *)
   CS ← IDT[Interrupt number * 4].selector;
   IP ← IDT[Interrupt number * 4].offset;

PROTECTED-MODE:
   Interrupt vector must be within IDT table limits,
      else #GP(vector number * 8+2+EXT);
   Descriptor AR byte must indicate interrupt gate, trap gate, or task gate,
      else #GP(vector number * 8+2+EXT);
   IF software interrupt (* i.e. caused by INT n, INT 3, or INTO *)
   THEN
      IF gate descriptor DPL < CPL
      THEN #GP(vector number * 8+2+EXT);
      FI;
   FI;
   Gate must be present, else #NP(vector number * 8+2+EXT);
   IF trap gate OR interrupt gate
   THEN GOTO TRAP-GATE-OR-INTERRUPT-GATE;
   ELSE GOTO TASK-GATE;
   FI;

TRAP-GATE-OR-INTERRUPT-GATE:
   Examine CS selector and descriptor given in the gate descriptor;
   Selector must be non-null, else #GP (EXT);
   Selector must be within its descriptor table limits
      ELSE #GP(selector+EXT);
   Descriptor AR byte must indicate code segment
      ELSE #GP(selector + EXT);
   Segment must be present, else #NP(selector+EXT);
   IF code segment is non-conforming AND DPL < CPL
   THEN GOTO INTERRUPT-TO-INNER-PRIVILEGE;
   ELSE
      IF code segment is conforming OR code segment DPL = CPL
      THEN GOTO INTERRUPT-TO-SAME-PRIVILEGE-LEVEL;
      ELSE #GP(CS selector + EXT);
      FI;
   FI;

INTERRUPT-TO-INNER-PRIVILEGE:
   Check selector and descriptor for new stack in current TSS;
      Selector must be non-null, else #GP(EXT);
      Selector index must be within its descriptor table limits
         ELSE #TS(SS selector+EXT);
      Selector's RPL must equal DPL of code segment, else #TS(SS
         selector+EXT);
      Stack segment DPL must equal DPL of code segment, else #TS(SS
         selector+EXT);
      Descriptor must indicate writable data segment, else #TS(SS
         selector+EXT);
      Segment must be present, else #SS(SS selector+EXT);
   IF 32-bit gate
   THEN New stack must have room for 20 bytes else #SS(0)
   ELSE New stack must have room for 10 bytes else #SS(0)
   FI;
   Instruction pointer must be within CS segment boundaries else #GP(0);
   Load new SS and eSP value from TSS;
   IF 32-bit gate
```

```
   THEN CS:EIP ← selector:offset from gate;
   ELSE CS:IP ← selector:offset from gate;
   FI;
   Load CS descriptor into invisible portion of CS register;
   Load SS descriptor into invisible portion of SS register;
   IF 32-bit gate
   THEN
      Push (long pointer to old stack) (* 3 words padded to 4 *);
      Push (EFLAGS);
      Push (long pointer to return location) (* 3 words padded to 4*);
   ELSE
      Push (long pointer to old stack) (* 2 words *);
      Push (FLAGS);
      Push (long pointer to return location) (* 2 words *);
   FI;
   Set CPL to new code segment DPL;
   Set RPL of CS to CPL;
   IF interrupt gate THEN IF ← 0 (* interrupt flag to 0 (disabled) *); FI;
   TF ← 0;
   NT ← 0;

INTERRUPT-FROM-V86-MODE:
   TempEFlags ← EFLAGS;
   VM ← 0;
   TF ← 0;
   IF service through Interrupt Gate THEN IF ← 0;
   TempSS ← SS;
   TempESP ← ESP;
   SS ← TSS.SS0; (* Change to level 0 stack segment *)
   ESP ← TSS.ESP0; (* Change to level 0 stack pointer *)
   Push(GS); (* padded to two words *)
   Push(FS); (* padded to two words *)
   Push(DS); (* padded to two words *)
   Push(ES); (* padded to two words *)
   GS ← 0;
   FS ← 0;
   DS ← 0;
   ES ← 0;
   Push(TempSS); (* padded to two words *)
   Push(TempESP);
   Push(TempEFlags);
   Push(CS); (* padded to two words *)
   Push(EIP);
   CS:EIP ← selector:offset from interrupt gate;
   (* Starts execution of new routine in 80386 Protected Mode *)

INTERRUPT-TO-SAME-PRIVILEGE-LEVEL:
   IF 32-bit gate
   THEN Current stack limits must allow pushing 10 bytes, else #SS(0);
   ELSE Current stack limits must allow pushing 6 bytes, else #SS(0);
   FI;
   IF interrupt was caused by exception with error code
   THEN Stack limits must allow push of two more bytes;
   ELSE #SS(0);
   FI;
   Instruction pointer must be in CS limit, else #GP(0);
   IF 32-bit gate
```

```
   THEN
      Push (EFLAGS);
      Push (long pointer to return location); (* 3 words padded to 4 *)
      CS:EIP ← selector:offset from gate;
   ELSE (* 16-bit gate *)
      Push (FLAGS);
      Push (long pointer to return location); (* 2 words *)
      CS:IP ← selector:offset from gate;
   FI;
   Load CS descriptor into invisible portion of CS register;
   Set the RPL field of CS to CPL;
   Push (error code); (* if any *)
   IF interrupt gate THEN IF ← 0; FI;
   TF ← 0;
   NT ← 0;

TASK-GATE:
   Examine selector to TSS, given in task gate descriptor;
      Must specify global in the local/global bit, else #TS(TSS selector);
      Index must be within GDT limits, else #TS(TSS selector);
      AR byte must specify available TSS (bottom bits 00001),
         else #TS(TSS selector;
      TSS must be present, else #NP(TSS selector);
   SWITCH-TASKS with nesting to TSS;
   IF interrupt was caused by fault with error code
   THEN
      Stack limits must allow push of two more bytes, else #SS(0);
      Push error code onto stack;
   FI;
   Instruction pointer must be in CS limit, else #GP(0);
```

Description

The INT instruction generates via software a call to an interrupt handler. The immediate operand, from 0 to 255, gives the index number into the Interrupt Descriptor Table (IDT) of the interrupt routine to be called. In Protected Mode, the IDT consists of an array of eight-byte descriptors; the descriptor for the interrupt invoked must indicate an interrupt, trap, or task gate. In Real Address Mode, the IDT is an array of four byte-long pointers. In Protected and Real Address Modes, the base linear address of the IDT is defined by the contents of the IDTR.

The INTO conditional software instruction is identical to the INT interrupt instruction except that the interrupt number is implicitly 4, and the interrupt is made only if the 80386 overflow flag is set.

The first 32 interrupts are reserved by Intel for system use. Some of these interrupts are use for internally generated exceptions.

INT n generally behaves like a far call except that the flags register is pushed onto the stack before the return address. Interrupt procedures return via the IRET instruction, which pops the flags and return address from the stack.

In Real Address Mode, INT n pushes the flags, CS, and the return IP onto the stack, in that order, then jumps to the long pointer indexed by the interrupt number.

Flags Affected

None

Protected Mode Exceptions

#GP, #NP, #SS, and #TS as indicated under "Operation" above

Real Address Mode Exceptions

None; if the SP or ESP = 1, 3, or 5 before executing INT or INTO,
the 80386 will shut down due to insufficient stack space

Virtual 8086 Mode Exceptions

#GP(0) fault if IOPL is less than 3, for INT only, to permit emulation;
Interrupt 3 (0CCH) generates Interrupt 3; INTO generates Interrupt 4
if the overflow flag equals 1

**IRET/IRETD —— Interrupt Return**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| CF | IRET | 22,pm=38 | Interrupt return (far return and pop flags) |
| CF | IRET | pm=82 | Interrupt return to lesser privilege |
| CF | IRET | ts | Interrupt return, different task (NT = 1) |
| CF | IRETD | 22,pm=38 | Interrupt return (far return and pop flags) |
| CF | IRETD | pm=82 | Interrupt return to lesser privilege |
| CF | IRETD | pm=60 | Interrupt return to V86 mode |
| CF | IRETD | ts | Interrupt return, different task (NT = 1) |

---

NOTE:
Values of ts are given by the following table:

| Old Task | New Task: 386 TSS VM = 0 | New Task: 386 TSS VM = 1 | New Task: 286 TSS |
|---|---|---|---|
| 386 TSS VM=0 | 275 | 224 | 271 |
| 286 TSS | 265 | 214 | 232 |

---

Operation

```
IF PE = 0
THEN (* Real-address mode *)
   IF OperandSize = 32 (* Instruction = IRETD *)
   THEN EIP ← Pop();
   ELSE (* Instruction = IRET *)
      IP ← Pop();
   FI;
   CS ← Pop();
   IF OperandSize = 32 (* Instruction = IRETD *)
   THEN EFLAGS ← Pop();
   ELSE (* Instruction = IRET *)
      FLAGS ← Pop();
   FI;
ELSE (* Protected mode *)
   IF VM = 1
   THEN #GP(0);
   ELSE
      IF NT = 1
      THEN GOTO TASK-RETURN;
      ELSE
         IF VM = 1 in flags image on stack
         THEN GO TO STACK-RETURN-TO-V86;
         ELSE GOTO STACK-RETURN;
         FI;
      FI;
   FI;
```

```
FI;STACK-RETURN-TO-V86: (* Interrupted procedure was in V86 mode *)
   IF return CS selector RPL < > 3
   THEN #GP(Return selector);
   FI;
   IF top 36 bytes of stack not within limits
   THEN #SS(0);
   FI;
   Examine return CS selector and associated descriptor:
      IF selector is null, THEN #GP(0); FI;
      IF selector index not within its descriptor table limits;
      THEN #GP(Return selector);
      FI;
      IF AR byte does not indicate code segment
      THEN #GP(Return selector);
      FI;
      IF code segment DPL not = 3;
      THEN #GP(Return selector);
      FI;
      IF code segment not present
      THEN #NP(Return selector);
      FI;

   Examine return SS selector and associated descriptor:
      IF selector is null THEN #GP(0); FI;
      IF selector index not within its descriptor table limits
      THEN #GP(SS selector);
      FI;
      IF selector RPL not = RPL of return CS selector
      THEN #GP(SS selector);
      FI;
      IF AR byte does not indicate a writable data segment
      THEN #GP(SS selector);
      FI;
      IF stack segment DPL not = RPL of return CS selector
      THEN #GP(SS selector);
      FI;
      IF SS not present
      THEN #NP(SS selector);
      FI;

   IF instruction pointer not within code segment limit  THEN #GP(0);
   FI;
   EFLAGS ← SS:[eSP + 8]; (* Sets VM in interrupted routine *)
   EIP ← Pop();
   CS ← Pop(); (* CS behaves as in 8086, due to VM = 1 *)
   throwaway ← Pop(); (* pop away EFLAGS already read *)
   ES ← Pop(); (* pop 2 words; throw away high-order word *)
   DS ← Pop(); (* pop 2 words; throw away high-order word *)
   FS ← Pop(); (* pop 2 words; throw away high-order word *)
   GS ← Pop(); (* pop 2 words; throw away high-order word *)
   IF CS.RPL > CPL
   THEN
      TempESP ← Pop();
      TempSS ← Pop();
      SS:ESP ← TempSS:TempESP;
   FI;
```

```
   (* Resume execution in Virtual 8086 mode *)

TASK-RETURN:
   Examine Back Link Selector in TSS addressed by the current task
      register:
      Must specify global in the local/global bit, else #TS(new TSS
         selector);
      Index must be within GDT limits, else #TS(new TSS selector);
      AR byte must specify TSS, else #TS(new TSS selector);
      New TSS must be busy, else #TS(new TSS selector);
      TSS must be present, else #NP(new TSS selector);
   SWITCH-TASKS without nesting to TSS specified by back link selector;
   Mark the task just abandoned as NOT BUSY;
   Instruction pointer must be within code segment limit ELSE #GP(0);

STACK-RETURN:
   IF OperandSize=32
   THEN Third word on stack must be within stack limits, else #SS(0);
   ELSE Second word on stack must be within stack limits, else #SS(0);
   FI;
   Return CS selector RPL must be ≥ CPL, else #GP(Return selector);
   IF return selector RPL = CPL
   THEN GOTO RETURN-SAME-LEVEL;
   ELSE GOTO RETURN-OUTER-LEVEL;
   FI;

RETURN-SAME-LEVEL:
   IF OperandSize=32
   THEN
      Top 12 bytes on stack must be within limits, else #SS(0);
      Return CS selector (at eSP+4) must be non-null, else #GP(0);
   ELSE
      Top 6 bytes on stack must be within limits, else #SS(0);
      Return CS selector (at eSP+2) must be non-null, else #GP(0);
   FI;
   Selector index must be within its descriptor table limits, else #GP
      (Return selector);
   AR byte must indicate code segment, else #GP(Return selector);
   IF non-conforming
   THEN code segment DPL must = CPL;
   ELSE #GP(Return selector);
   FI;
   IF conforming
   THEN code segment DPL must be ≤ CPL, else #GP(Return selector);
   Segment must be present, else #NP(Return selector);
   Instruction pointer must be within code segment boundaries, else #GP(0);
   FI;
   IF OperandSize=32
   THEN
      Load CS:EIP from stack;
      Load CS-register with new code segment descriptor;
      Load EFLAGS with third doubleword from stack;
      Increment eSP by 12;
   ELSE
      Load CS-register with new code segment descriptor;
      Load FLAGS with third word on stack;
      Increment eSP by 6;
```

```
   FI;

RETURN-OUTER-LEVEL:
   IF OperandSize=32
   THEN Top 20 bytes on stack must be within limits, else #SS(0);
   ELSE Top 10 bytes on stack must be within limits, else #SS(0);
   FI;
   Examine return CS selector and associated descriptor:
      Selector must be non-null, else #GP(0);
      Selector index must be within its descriptor table limits;
         ELSE #GP(Return selector);
      AR byte must indicate code segment, else #GP(Return selector);
      IF non-conforming
      THEN code segment DPL must = CS selector RPL;
      ELSE #GP(Return selector);
      FI;
      IF conforming
      THEN code segment DPL must be > CPL;
      ELSE #GP(Return selector);
      FI;
      Segment must be present, else #NP(Return selector);
   Examine return SS selector and associated descriptor:
      Selector must be non-null, else #GP(0);
      Selector index must be within its descriptor table limits
         ELSE #GP(SS selector);
      Selector RPL must equal the RPL of the return CS selector
         ELSE #GP(SS selector);
      AR byte must indicate a writable data segment, else #GP(SS selector);
      Stack segment DPL must equal the RPL of the return CS selector
         ELSE #GP(SS selector);
      SS must be present, else #NP(SS selector);

   Instruction pointer must be within code segment limit ELSE #GP(0);
   IF OperandSize=32
   THEN
      Load CS:EIP from stack;
      Load EFLAGS with values at (eSP+8);
   ELSE
      Load CS:IP from stack;
      Load FLAGS with values at (eSP+4);
   FI;
   Load SS:eSP from stack;
   Set CPL to the RPL of the return CS selector;
   Load the CS register with the CS descriptor;
   Load the SS register with the SS descriptor;
   FOR each of ES, FS, GS, and DS
   DO;
      IF the current value of the register is not valid for the outer level;
      THEN zero the register and clear the valid flag;
      FI;
      To be valid, the register setting must satisfy the following
         properties:
         Selector index must be within descriptor table limits;
         AR byte must indicate data or readable code segment;
         IF segment is data or non-conforming code,
         THEN DPL must be ≥ CPL, or DPL must be ≥ RPL;
   OD;
```

Description

In Real Address Mode, IRET pops the instruction pointer, CS, and the flags register from the stack and resumes the interrupted routine.

In Protected Mode, the action of IRET depends on the setting of the nested task flag (NT) bit in the flag register. When popping the new flag image from the stack, the IOPL bits in the flag register are changed only when CPL equals 0.

If NT equals 0, IRET returns from an interrupt procedure without a task switch. The code returned to must be equally or less privileged than the interrupt routine (as indicated by the RPL bits of the CS selector popped from the stack). If the destination code is less privileged, IRET also pops the stack pointer and SS from the stack.

If NT equals 1, IRET reverses the operation of a CALL or INT that caused a task switch. The updated state of the task executing IRET is saved in its task state segment. If the task is reentered later, the code that follows IRET is executed.

Flags Affected

All; the flags register is popped from stack

Protected Mode Exceptions

#GP, #NP, or #SS, as indicated under "Operation" above

Real Address Mode Exceptions

Interrupt 13 if any part of the operand being popped lies beyond address 0FFFFH

Virtual 8086 Mode Exceptions

#GP(0) fault if IOPL is less than 3, to permit emulation

## Jcc — Jump if Condition is Met

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 77 cb | JA rel8 | 7+m,3 | Jump short if above (CF=0 and ZF=0) |
| 73 cb | JAE rel8 | 7+m,3 | Jump short if above or equal (CF=0) |
| 72 cb | JB rel8 | 7+m,3 | Jump short if below (CF=1) |
| 76 cb | JBE rel8 | 7+m,3 | Jump short if below or (CF=1 or ZF=1) |
| 72 cb | JC rel8 | 7+m,3 | Jump short if carry (CF=1) |
| E3 cb | JCXZ rel8 | 9+m,5 | Jump short if CX register is 0 |
| E3 cb | JECXZ rel8 | 9+m,5 | Jump short if ECX register is 0 |
| 74 cb | JE rel8 | 7+m,3 | Jump short if equal (ZF=1) |
| 74 cb | JZ rel8 | 7+m,3 | Jump short if 0 (ZF=1) |
| 7F cb | JG rel8 | 7+m,3 | Jump short if greater (ZF=0 and SF=OF) |
| 7D cb | JGE rel8 | 7+m,3 | Jump short if greater or equal (SF=OF) |
| 7C cb | JL rel8 | 7+m,3 | Jump short if less (SF≠OF) |
| 7E cb | JLE rel8 | 7+m,3 | Jump short if less or equal (ZF=1 and SF≠OF) |
| 76 cb | JNA rel8 | 7+m,3 | Jump short if not above (CF=1 ZF=1) |
| 72 cb | JNAE rel8 | 7+m,3 | Jump short if not above or equal (CF=1) |
| 73 cb | JNB rel8 | 7+m,3 | Jump short if not below (CF=0) |
| 77 cb | JNBE rel8 | 7+m,3 | Jump short if not below or equal (CF=0 and ZF=0) |
| 73 cb | JNC rel8 | 7+m,3 | Jump short if not carry (CF=0) |
| 75 cb | JNE rel8 | 7+m,3 | Jump short if not equal (ZF=0) |
| 7E cb | JNG rel8 | 7+m,3 | Jump short if not greater (ZF=1 or SF≠OF) |
| 7C cb | JNGE rel8 | 7+m,3 | Jump short if not greater or equal (SF≠OF) |
| 7D cb | JNL rel8 | 7+m,3 | Jump short if not less (SF=OF) |
| 7F cb | JNLE rel8 | 7+m,3 | Jump short if not less or equal (ZF=0 and SF=OF) |
| 71 cb | JNO rel8 | 7+m,3 | Jump short if not overflow (OF=0) |
| 7B cb | JNP rel8 | 7+m,3 | Jump short if not parity (PF=0) |
| 79 cb | JNS rel8 | 7+m,3 | Jump short if not sign (SF=0) |
| 75 cb | JNZ rel8 | 7+m,3 | Jump short if not zero (ZF=0) |
| 70 cb | JO rel8 | 7+m,3 | Jump short if overflow (OF=1) |
| 7A cb | JP rel8 | 7+m,3 | Jump short if parity (PF=1) |
| 7A cb | JPE rel8 | 7+m,3 | Jump short if parity even (PF=1) |
| 7B cb | JPO rel8 | 7+m,3 | Jump short if parity odd (PF=0) |
| 78 cb | JS rel8 | 7+m,3 | Jump short if sign (SF=1) |
| 74 cb | JZ rel8 | 7+m,3 | Jump short if zero (ZF = 1) |
| 0F 87 cw/cd | JA rel16/32 | 7+m,3 | Jump near if above (CF=0 and ZF=0) |
| 0F 83 cw/cd | JAE rel16/32 | 7+m,3 | Jump near if above or equal (CF=0) |
| 0F 82 cw/cd | JB rel16/32 | 7+m,3 | Jump near if below (CF=1) |
| 0F 86 cw/cd | JBE rel16/32 | 7+m,3 | Jump near if below or equal (CF=1 or ZF=1) |
| 0F 82 cw/cd | JC rel16/32 | 7+m,3 | Jump near if carry (CF=1) |
| 0F 84 cw/cd | JE rel16/32 | 7+m,3 | Jump near if equal (ZF=1) |
| 0F 84 cw/cd | JZ rel16/32 | 7+m,3 | Jump near if 0 (ZF=1) |
| 0F 8F cw/cd | JG rel16/32 | 7+m,3 | Jump near if greater (ZF=0 and SF=OF) |
| 0F 8D cw/cd | JGE rel16/32 | 7+m,3 | Jump near if greater or equal (SF=OF) |
| 0F 8C cw/cd | JL rel16/32 | 7+m,3 | Jump near if less (SF≠OF) |
| 0F 8E cw/cd | JLE rel16/32 | 7+m,3 | Jump near if less or equal (ZF=1 and SF≠OF) |
| 0F 86 cw/cd | JNA rel16/32 | 7+m,3 | Jump near if not above (CF=1 or ZF=1) |
| 0F 82 cw/cd | JNAE rel16/32 | 7+m,3 | Jump near if not above or equal (CF=1) |
| 0F 83 cw/cd | JNB rel16/32 | 7+m,3 | Jump near if not below (CF=0) |
| 0F 87 cw/cd | JNBE rel16/32 | 7+m,3 | Jump near if not below or equal (CF=0 and ZF=0) |
| 0F 83 cw/cd | JNC rel16/32 | 7+m,3 | Jump near if not carry (CF=0) |
| 0F 85 cw/cd | JNE rel16/32 | 7+m,3 | Jump near if not equal (ZF=0) |
| 0F 8E cw/cd | JNG rel16/32 | 7+m,3 | Jump near if not greater (ZF=1 or SF≠OF) |
| 0F 8C cw/cd | JNGE rel16/32 | 7+m,3 | Jump near if not greater or equal (SF≠OF) |
| 0F 8D cw/cd | JNL rel16/32 | 7+m,3 | Jump near if not less (SF=OF) |
| 0F 8F cw/cd | JNLE rel16/32 | 7+m,3 | Jump near if not less or equal (ZF=0 and SF=OF) |
| 0F 81 cw/cd | JNO rel16/32 | 7+m,3 | Jump near if not overflow (OF=0) |
| 0F 8B cw/cd | JNP rel16/32 | 7+m,3 | Jump near if not parity (PF=0) |
| 0F 89 cw/cd | JNS rel16/32 | 7+m,3 | Jump near if not sign (SF=0) |
| 0F 85 cw/cd | JNZ rel16/32 | 7+m,3 | Jump near if not zero (ZF=0) |
| 0F 80 cw/cd | JO rel16/32 | 7+m,3 | Jump near if overflow (OF=1) |
| 0F 8A cw/cd | JP rel16/32 | 7+m,3 | Jump near if parity (PF=1) |
| 0F 8A cw/cd | JPE rel16/32 | 7+m,3 | Jump near if parity even (PF=1) |
| 0F 8B cw/cd | JPO rel16/32 | 7+m,3 | Jump near if parity odd (PF=0) |
| 0F 88 cw/cd | JS rel16/32 | 7+m,3 | Jump near if sign (SF=1) |
| 0F 84 cw/cd | JZ rel16/32 | 7+m,3 | Jump near if 0 (ZF=1) |

NOTES:
The first clock count is for the true condition (branch taken); the second clock count is for the false condition (branch not taken). rel16/32 indicates that these instructions map to two; one with a 16-bit relative displacement, the other with a 32-bit relative displacement, depending on the operand-size attribute of the instruction.

Operation

```
IF condition
THEN
   EIP ← EIP + SignExtend(rel8/16/32);
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFH;
   FI;
FI;
```

Description

Conditional jumps (except JCXZ) test the flags which have been set by a previous instruction. The conditions for each mnemonic are given in parentheses after each description above. The terms "less" and "greater" are used for comparisons of signed integers; "above" and "below" are used for unsigned integers.

If the given condition is true, a jump is made to the location provided as the operand. Instruction coding is most efficient when the target for the conditional jump is in the current code segment and within -128 to +127 bytes of the next instruction's first byte. The jump can also target -32768 thru +32767 (segment size attribute 16) or $-2^{31}$ thru $+2^{31}-1$ (segment size attribute 32) relative to the next instruction's first byte. When the target for the conditional jump is in a different segment, use the opposite case of the jump instruction (i.e., JE and JNE), and then access the target with an unconditional far jump to the other segment. For example, you cannot code——

```
JZ FARLABEL;
```

You must instead code——

```
   JNZ BEYOND;
   JMP FARLABEL;
BEYOND:
```

Because there can be several ways to interpret a particular state of the flags, ASM386 provides more than one mnemonic for most of the conditional jump opcodes. For example, if you compared two characters in AX and want to jump if they are equal, use JE; or, if you ANDed AX with a bit field mask and only want to jump if the result is 0, use JZ, a synonym for JE.

JCXZ differs from other conditional jumps because it tests the contents of the CX or ECX register for 0, not the flags. JCXZ is useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as LOOPNE TARGET LABEL. The JCXZ prevents entering the loop with CX or ECX equal to zero, which would cause the loop to execute 64K or 32G times instead of zero times.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if the offset jumped to is beyond the limits of the code segment

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

**JMP — Jump**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| EB cb | JMP rel8 | 7+m | Jump short |
| E9 cw | JMP rel16 | 7+m | Jump near, displacement relative to next instruction |
| FF /4 | JMP r/m16 | 7+m/10+m | Jump near indirect |
| EA cd | JMP ptr16:16 | 12+m,pm=27+m | Jump intersegment, 4-byte immediate address |
| EA cd | JMP ptr16:16 | pm=45+m | Jump to call gate, same privilege |
| EA cd | JMP ptr16:16 | ts | Jump via task state segment |
| EA cd | JMP ptr16:16 | ts | Jump via task gate |
| FF /5 | JMP m16:16 | 43+m,pm=31+m | Jump r/m16:16 indirect and intersegment |
| FF /5 | JMP m16:16 | pm=49+m | Jump to call gate, same privilege |
| FF /5 | JMP m16:16 | 5 + ts | Jump via task state segment |
| FF /5 | JMP m16:16 | 5 + ts | Jump via task gate |
| E9 cd | JMP rel32 | 7+m | Jump near, displacement relative to next instruction |
| FF /4 | JMP r/m32 | 7+m,10+m | Jump near, indirect |
| EA cp | JMP ptr16:32 | 12+m,pm=27+m | Jump intersegment, 6-byte immediate address |
| EA cp | JMP ptr16:32 | pm=45+m | Jump to call gate, same privilege |
| EA cp | JMP ptr16:32 | ts | Jump via task state segment |
| EA cp | JMP ptr16:32 | ts | Jump via task gate |
| FF /5 | JMP m16:32 | 43+m,pm=31+m | Jump intersegment, address at r/m dword |
| FF /5 | JMP m16:32 | pm=49+m | Jump to call gate, same privilege |
| FF /5 | JMP m16:32 | 5 + ts | Jump via task state segment |
| FF /5 | JMP m16:32 | 5 + ts | Jump via task gate |

---

NOTE:
Values of ts are given by the following table:

| | New Task | | | | |
|---|---|---|---|---|---|
| | 386 TSS VM = 0 | | 386 TASK VM = 1 | | 286 TSS |
| Old Task | Via Task Gate? | | | | |
| | N | Y | N | Y | N | Y |
| 386 TSS VM=0 | 303 | 312 | 220 | 229 | 276 | 285 |
| 286 TSS | 301 | 310 | 218 | 227 | 274 | 283 |

---

Operation

```
IF instruction = relative JMP
   (* i.e. operand is rel8, rel16, or rel32 *)
THEN
   EIP ← EIP + rel8/16/32;
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFH;
   FI;
FI;
IF instruction = near indirect JMP
   (* i.e. operand is r/m16 or r/m32 *)
THEN
   IF OperandSize = 16
```

```
   THEN
      EIP ← [r/m16] AND 0000FFFFH;
   ELSE (* OperandSize = 32 *)
      EIP ← [r/m32];
   FI;
FI;

IF (PE = 0 OR (PE = 1 AND VM = 1)) (* real mode or V86 mode *)
   AND instruction = far JMP
   (* i.e., operand type is m16:16, m16:32, ptr16:16, ptr16:32 *)
THEN GOTO REAL-OR-V86-MODE;
   IF operand type = m16:16 or m16:32
   THEN (* indirect *)
      IF OperandSize = 16
      THEN
         CS:IP ← [m16:16];
         EIP ← EIP AND 0000FFFFH; (* clear upper 16 bits *)
      ELSE (* OperandSize = 32 *)
         CS:EIP ← [m16:32];
      FI;
   FI;
   IF operand type = ptr16:16 or ptr16:32
   THEN
      IF OperandSize = 16
      THEN
         CS:IP ← ptr16:16;
         EIP ← EIP AND 0000FFFFH; (* clear upper 16 bits *)
      ELSE (* OperandSize = 32 *)
         CS:EIP ← ptr16:32;
      FI;
   FI;
FI;

IF (PE = 1 AND VM = 0) (* Protected mode, not V86 mode *)
   AND instruction = far JMP
THEN
   IF operand type = m16:16 or m16:32
   THEN (* indirect *)
      check access of EA dword;
      #GP(0) or #SS(0) IF limit violation;
   FI;
   Destination selector is not null ELSE #GP(0)
   Destination selector index is within its descriptor table limits ELSE
#GP(selector)
   Depending on AR byte of destination descriptor:
      GOTO CONFORMING-CODE-SEGMENT;
      GOTO NONCONFORMING-CODE-SEGMENT;
      GOTO CALL-GATE;
      GOTO TASK-GATE;
      GOTO TASK-STATE-SEGMENT;
   ELSE #GP(selector); (* illegal AR byte in descriptor *)
FI;

CONFORMING-CODE-SEGMENT:
   Descriptor DPL must be ≤ CPL ELSE #GP(selector);
   Segment must be present ELSE #NP(selector);
   Instruction pointer must be within code-segment limit ELSE #GP(0);
```

```
   IF OperandSize = 32
   THEN Load CS:EIP from destination pointer;
   ELSE Load CS:IP from destination pointer;
   FI;
   Load CS register with new segment descriptor;

NONCONFORMING-CODE-SEGMENT:
   RPL of destination selector must be ≤ CPL ELSE #GP(selector);
   Descriptor DPL must be = CPL ELSE #GP(selector);
   Segment must be present ELSE # NP(selector);
   Instruction pointer must be within code-segment limit ELSE #GP(0);
   IF OperandSize = 32
   THEN Load CS:EIP from destination pointer;
   ELSE Load CS:IP from destination pointer;
   FI;
   Load CS register with new segment descriptor;
   Set RPL field of CS register to CPL;

CALL-GATE:
   Descriptor DPL must be ≥ CPL ELSE #GP(gate selector);
   Descriptor DPL must be ≥ gate selector RPL ELSE #GP(gate selector);
   Gate must be present ELSE #NP(gate selector);
   Examine selector to code segment given in call gate descriptor:
      Selector must not be null ELSE #GP(0);
      Selector must be within its descriptor table limits ELSE
         #GP(CS selector);
      Descriptor AR byte must indicate code segment
         ELSE #GP(CS selector);
      IF non-conforming
      THEN code-segment descriptor, DPL must = CPL
      ELSE #GP(CS selector);
      FI;
      IF conforming
      THEN code-segment descriptor DPL must be ≤ CPL;
      ELSE #GP(CS selector);
      Code segment must be present ELSE #NP(CS selector);
      Instruction pointer must be within code-segment limit ELSE #GP(0);
      IF OperandSize = 32
      THEN Load CS:EIP from call gate;
      ELSE Load CS:IP from call gate;
      FI;
   Load CS register with new code-segment descriptor;
   Set RPL of CS to CPL

TASK-GATE:
   Gate descriptor DPL must be ≥ CPL ELSE #GP(gate selector);
   Gate descriptor DPL must be ≥ gate selector RPL ELSE #GP(gate
     selector);
   Task Gate must be present ELSE #NP(gate selector);
   Examine selector to TSS, given in Task Gate descriptor:
   Must specify global in the local/global bit ELSE #GP(TSS selector);
   Index must be within GDT limits ELSE #GP(TSS selector);
   Descriptor AR byte must specify available TSS (bottom bits 00001);
      ELSE #GP(TSS selector);
   Task State Segment must be present ELSE #NP(TSS selector);
SWITCH-TASKS (without nesting) to TSS;
Instruction pointer must be within code-segment limit ELSE #GP(0);
```

```
TASK-STATE-SEGMENT:
   TSS DPL must be ≥ CPL ELSE #GP(TSS selector);
   TSS DPL must be ≥ TSS selector RPL ELSE #GP(TSS selector);
   Descriptor AR byte must specify available TSS (bottom bits 00001)
      ELSE #GP(TSS selector);
   Task State Segment must be present ELSE #NP(TSS selector);
   SWITCH-TASKS (without nesting) to TSS;
   Instruction pointer must be within code-segment limit ELSE #GP(0);
```

Description

The JMP instruction transfers control to a different point in the instruction stream without recording return information.

The action of the various forms of the instruction are shown below.

Jumps with destinations of type r/m16, r/m32, rel16, and rel32 are near jumps and do not involve changing the segment register value.

The JMP rel16 and JMP rel32 forms of the instruction add an offset to the address of the instruction following the JMP to determine the destination. The rel16 form is used when the instruction's operand-size attribute is 16 bits (segment size attribute 16 only); rel32 is used when the operand-size attribute is 32 bits (segment size attribute 32 only). The result is stored in the 32-bit EIP register. With rel16, the upper 16 bits of EIP are cleared, which results in an offset whose value does not exceed 16 bits.

JMP r/m16 and JMP r/m32 specifies a register or memory location from which the absolute offset from the procedure is fetched. The offset fetched from r/m is 32 bits for an operand-size attribute of 32 bits (r/m32), or 16 bits for an operand-size attribute of 16 bits (r/m16).

The JMP ptr16:16 and ptr16:32 forms of the instruction use a four-byte or six-byte operand as a long pointer to the destination. The JMP and  forms fetch the long pointer from the memory location specified (indirection). In Real Address Mode or Virtual 8086 Mode, the long pointer provides 16 bits for the CS register and 16 or 32 bits for the EIP register (depending on the operand-size attribute). In Protected Mode, both long pointer forms consult the Access Rights (AR) byte in the descriptor indexed by the selector part of the long pointer.

Depending on the value of the AR byte, the jump will perform one of the following types of control transfers:

- A jump to a code segment at the same privilege level
- A task switch

For more information on protected mode control transfers, refer to Chapter 6 and Chapter 7.

Flags Affected

All if a task switch takes place; none if no task switch occurs

Protected Mode Exceptions

Far jumps: #GP, #NP, #SS, and #TS, as indicated in the list above.

Near direct jumps: #GP(0) if procedure location is beyond the code segment limits.

Near indirect jumps: #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments: #SS(0) for an illegal address in the SS segment; #GP if the indirect offset obtained is beyond the code segment limits; #PF(fault-code) for a page fault.

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would be outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as under Real Address Mode; #PF(fault-code) for a page fault

## LAHF —— Load Flags into AH Register

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 9F | LAHF | 2 | Load: AH = flags SF ZF xx AF xx PF xx CF |

### Operation

AH ← SF:ZF:xx:AF:xx:PF:xx:CF;

### Description

LAHF transfers the low byte of the flags word to AH. The bits, from MSB to LSB, are sign, zero, indeterminate, auxiliary, carry, indeterminate, parity, indeterminate, and carry.

### Flags Affected

None

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

**LAR —— Load Access Rights Byte**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 02 /r | LAR r16,r/m16 | pm=15/16 | r16 ← r/m16 masked by FF00 |
| 0F 02 /r | LAR r32,r/m32 | pm=15/16 | r32 ← r/m32 masked by 00FxFF00 |

Description

The LAR instruction stores a marked form of the second doubleword of the descriptor for the source selector if the selector is visible at the CPL (modified by the selector's RPL) and is a valid descriptor type. The destination register is loaded with the high-order doubleword of the descriptor masked by 00FxFF00, and ZF is set to 1. The x indicates that the four bits corresponding to the upper four bits of the limit are undefined in the value loaded by LAR. If the selector is invisible or of the wrong type, ZF is cleared.

If the 32-bit operand size is specified, the entire 32-bit value is loaded into the 32-bit destination register. If the 16-bit operand size is specified, the lower 16-bits of this value are stored in the 16-bit destination register.

All code and data segment descriptors are valid for LAR.

The valid special segment and gate descriptor types for LAR are given in the following table:

| Type | Name | Valid/Invalid |
|---|---|---|
| 0 | Invalid | Invalid |
| 1 | Available 80286 TSS | Valid |
| 2 | LDT | Valid |
| 3 | Busy 80286 TSS | Valid |
| 4 | 80286 call gate | Valid |
| 5 | 80286/80386 task gate | Valid |
| 6 | 80286 trap gate | Valid |
| 7 | 80286 interrupt gate | Valid |
| 8 | Invalid | Invalid |
| 9 | Available 80386 TSS | Valid |
| A | Invalid | Invalid |
| B | Busy 80386 TSS | Valid |
| C | 80386 call gate | Valid |
| D | Invalid | Invalid |
| E | 80386 trap gate | Valid |
| F | 80386 interrupt gate | Valid |

Flags Affected

ZF as described above

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 6; LAR is unrecognized in Real Address Mode

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode

**LEA —— Load Effective Address**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 8D /r | LEA r16,m | 2 | Store effective address for m in register r16 |
| 8D /r | LEA r32,m | 2 | Store effective address for m in register r32 |
| 8D /r | LEA r16,m | 2 | Store effective address for m in register r16 |
| 8D /r | LEA r32,m | 2 | Store effective address for m in register r32 |

Operation

```
IF OperandSize = 16 AND AddressSize = 16
THEN r16 ← Addr(m);
ELSE
   IF OperandSize = 16 AND AddressSize = 32
   THEN
      r16 ← Truncate_to_16bits(Addr(m));   (* 32-bit address *)
   ELSE
      IF OperandSize = 32 AND AddressSize = 16
      THEN
         r32 ← Truncate_to_16bits(Addr(m));
      ELSE
         IF OperandSize = 32 AND AddressSize = 32
         THEN  r32 ← Addr(m);
         FI;
      FI;
   FI;
FI;
```

Description

LEA calculates the effective address (offset part) and stores it in the specified register. The operand-size attribute of the instruction (represented by OperandSize in the algorithm under "Operation" above) is determined by the chosen register. The address-size attribute (represented by AddressSize) is determined by the USE attribute of the segment containing the second operand. The address-size and operand-size attributes affect the action performed by LEA, as follows:

| Operand Size | Address Size | Action Performed |
|---|---|---|
| 16 | 16 | 16-bit effective address is calculated and stored in requested 16-bit register destination. |
| 16 | 32 | 32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination. |
| 32 | 16 | 16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination. |
| 32 | 32 | 32-bit effective address is calculated and stored in the requested 32-bit register destination. |

Flags Affected

None

Protected Mode Exceptions

#UD if the second operand is a register

Real Address Mode Exceptions

Interrupt 6 if the second operand is a register

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode

**LEAVE —— High Level Procedure Exit**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| C9 | LEAVE | 4 | Set SP to BP, then pop BP |
| C9 | LEAVE | 4 | Set ESP to EBP, then pop EBP |

Operation

```
IF StackAddrSize = 16
THEN
   SP ← BP;
ELSE (* StackAddrSize = 32 *)
   ESP ← EBP;
FI;
IF OperandSize = 16
THEN
   BP ← Pop();
ELSE (* OperandSize = 32 *)
   EBP ← Pop();
FI;
```

Description

LEAVE reverses the actions of the ENTER instruction. By copying the frame pointer to the stack pointer, LEAVE releases the stack space used by a procedure for its local variables. The old frame pointer is popped into BP or EBP, restoring the caller's frame. A subsequent RET instruction removes any arguments pushed onto the stack of the exiting procedure.

Flags Affected

None

Protected Mode Exceptions

#SS(0) if BP does not point to a location within the limits of the current stack segment

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode

## LGDT/LIDT —— Load Global/Interrupt Descriptor Table Register

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 01 /2 | LGDT m16&32 | 11 | Load m into GDTR |
| 0F 01 /3 | LIDT m16&32 | 11 | Load m into IDTR |

Operation

```
IF instruction = LIDT
THEN
   IF OperandSize = 16
   THEN IDTR.Limit:Base ← m16:24 (* 24 bits of base loaded *)
   ELSE IDTR.Limit:Base ← m16:32
   FI;
ELSE (* instruction = LGDT *)
   IF OperandSize = 16
   THEN GDTR.Limit:Base ← m16:24 (* 24 bits of base loaded *)
   ELSE GDTR.Limit:Base ← m16:32;
   FI;
FI;
```

Description

The LGDT and LIDT instructions load a linear base address and limit value from a six-byte data operand in memory into the GDTR or IDTR, respectively. If a 16-bit operand is used with LGDT or LIDT, the register is loaded with a 16-bit limit and a 24-bit base, and the high-order eight bits of the six-byte data operand are not used. If a 32-bit operand is used, a 16-bit limit and a 32-bit base is loaded; the high-order eight bits of the six-byte operand are used as high-order base address bits.

The SGDT and SIDT instructions always store into all 48 bits of the six-byte data operand. With the 80286, the upper eight bits are undefined after SGDT or SIDT is executed. With the 80386, the upper eight bits are written with the high-order eight address bits, for both a 16-bit operand and a 32-bit operand. If LGDT or LIDT is used with a 16-bit operand to load the register stored by SGDT or SIDT, the upper eight bits are stored as zeros.

LGDT and LIDT appear in operating system software; they are not used in application programs. They are the only instructions that directly load a linear address (i.e., not a segment relative address) in 80386 Protected Mode.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if the current privilege level is not 0; #UD if the source operand is a register; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; Interrupt 6 if the source operand is a register

---

Note:
These instructions are valid in Real Address Mode to allow power-up initialization for Protected Mode

---

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**LGS/LSS/LDS/LES/LFS —— Load Full Pointer**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| C5 /r | LDS r16,m16:16 | 7,p=22 | Load DS:r16 with pointer from memory |
| C5 /r | LDS r32,m16:32 | 7,p=22 | Load DS:r32 with pointer from memory |
| 0F B2 /r | LSS r16,m16:16 | 7,p=22 | Load SS:r16 with pointer from memory |
| 0F B2 /r | LSS r32,m16:32 | 7,p=22 | Load SS:r32 with pointer from memory |
| C4 /r | LES r16,m16:16 | 7,p=22 | Load ES:r16 with pointer from memory |
| C4 /r | LES r32,m16:32 | 7,p=22 | Load ES:r32 with pointer from memory |
| 0F B4 /r | LFS r16,m16:16 | 7,p=25 | Load FS:r16 with pointer from memory |
| 0F B4 /r | LFS r32,m16:32 | 7,p=25 | Load FS:r32 with pointer from memory |
| 0F B5 /r | LGS r16,m16:16 | 7,p=25 | Load GS:r16 with pointer from memory |
| 0F B5 /r | LGS r32,m16:32 | 7,p=25 | Load GS:r32 with pointer from memory |

Operation

```
CASE instruction OF
   LSS: Sreg is SS; (* Load SS register *)
   LDS: Sreg is DS; (* Load DS register *)
   LES: Sreg is ES; (* Load ES register *)
   LFS: Sreg is FS; (* Load FS register *)
   LGS: Sreg is DS; (* Load GS register *)
ESAC;
IF (OperandSize = 16)
THEN
   r16 ← [Effective Address]; (* 16-bit transfer *)
   Sreg ← [Effective Address + 2]; (* 16-bit transfer *)
   (* In Protected Mode, load the descriptor into the segment register *)
ELSE (* OperandSize = 32 *)
   r32 ← [Effective Address]; (* 32-bit transfer *)
   Sreg ← [Effective Address + 4]; (* 16-bit transfer *)
   (* In Protected Mode, load the descriptor into the segment register *)
FI;
```

Description

These instructions read a full pointer from memory and store it in the selected segment register:register pair. The full pointer loads 16 bits into the segment register SS, DS, ES, FS, or GS. The other register loads 32 bits if the operand-size attribute is 32 bits, or loads 16 bits if the operand-size attribute is 16 bits. The other 16- or 32-bit register to be loaded is determined by the r16 or r32 register operand specified.

When an assignment is made to one of the segment registers, the descriptor is also loaded into the segment register. The data for the register is obtained from the descriptor table entry for the selector given.

A null selector (values 0000-0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a null selector to address memory causes a #GP(0) exception. No memory reference to the segment occurs.)

The following is a listing of the Protected Mode checks and actions taken in the loading of a segment register:

```
IF SS is loaded:
   IF selector is null THEN #GP(0); FI;
   Selector index must be within its descriptor table limits ELSE
      #GP(selector);
   Selector's RPL must equal CPL ELSE #GP(selector);
   AR byte must indicate a writable data segment ELSE #GP(selector);
   DPL in the AR byte must equal CPL ELSE #GP(selector);
   Segment must be marked present ELSE #SS(selector);
   Load SS with selector;
   Load SS with descriptor;
IF DS, ES, FS, or GS is loaded with non-null selector:
   Selector index must be within its descriptor table limits ELSE
      #GP(selector);
   AR byte must indicate data or readable code segment ELSE
      #GP(selector);
   IF data or nonconforming code
   THEN both the RPL and the CPL must be less than or equal to DPL in
      AR byte;
   ELSE #GP(selector);
   Segment must be marked present ELSE #NP(selector);
Load segment register with selector and RPL bits;
Load segment register with descriptor;
IF DS, ES, FS or GS is loaded with a null selector:
   Clear descriptor valid bit;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; the second operand must be a memory operand, not a register; #GP(0) if a null selector is loaded into SS; #PF(fault-code) for a page fault

Real Address Mode Exceptions

The second operand must be a memory operand, not a register; Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## LLDT —— Load Local Descriptor Table Register

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 00 /2 | LLDT r/m16 | 20 | Load selector r/m16 into LDTR |

Operation

LDTR ← SRC;

Description

LLDT loads the Local Descriptor Table register (LDTR). The word operand (memory or register) to LLDT should contain a selector to the Global Descriptor Table (GDT). The GDT entry should be a Local Descriptor Table. If so, then the LDTR is loaded from the entry. The descriptor registers DS, ES, SS, FS, GS, and CS are not affected. The LDT field in the task state segment does not change.

The selector operand can be 0; if so, the LDTR is marked invalid. All descriptor references (except by the LAR, VERR, VERW or LSL instructions) cause a #GP fault.

LLDT is used in operating system software; it is not used in application programs.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if the current privilege level is not 0; #GP(selector) if the selector operand does not point into the Global Descriptor Table, or if the entry in the GDT is not a Local Descriptor Table; #NP(selector) if the LDT descriptor is not present; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 6; LLDT is not recognized in Real Address Mode

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode (because the instruction is not recognized, it will not execute or perform a memory reference)

Note

The operand-size attribute has no effect on this instruction.

**LMSW —— Load Machine Status Word**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 01 /6 | LMSW r/m16 | 10/13 | Load r/m16 in machine status word |

Operation

MSW ← r/m16; (* 16 bits is stored in the machine status word *)

Description

LMSW loads the machine status word (part of CR0) from the source operand. This instruction can be used to switch to Protected Mode; if so, it must be followed by an intrasegment jump to flush the instruction queue. LMSW will not switch back to Real Address Mode.

LMSW is used only in operating system software. It is not used in application programs.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if the current privilege level is not 0; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

Notes

The operand-size attribute has no effect on this instruction. This instruction is provided for compatibility with the 80286; 80386 programs should use MOV CR0, ... instead.

## LOCK —— Assert LOCK# Signal Prefix

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F0 | LOCK | 0 | Assert LOCK# signal for the next instruction |

### Description

The LOCK prefix causes the LOCK# signal of the 80386 to be asserted during execution of the instruction that follows it. In a multiprocessor environment, this signal can be used to ensure that the 80386 has exclusive use of any shared memory while LOCK# is asserted. The read-modify-write sequence typically used to implement test-and-set on the 80386 is the BTS instruction.

The LOCK prefix functions only with the following instructions:

| | |
|---|---|
| BT, BTS, BTR, BTC | mem, reg/imm |
| XCHG | reg, mem |
| XCHG | mem, reg |
| ADD, OR, ADC, SBB, AND, SUB, XOR | mem, reg/imm |
| NOT, NEG, INC, DEC | mem |

An undefined opcode trap will be generated if a LOCK prefix is used with any instruction not listed above.

XCHG always asserts LOCK# regardless of the presence or absence of the LOCK prefix.

The integrity of the LOCK is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

Locked access is not assured if another 80386 processor is executing an instruction concurrently that has one of the following characteristics:

- Is not preceded by a LOCK prefix
- Is not one of the instructions in the preceding list
- Specifies a memory operand that does not exactly overlap the destination operand. Locking is not guaranteed for partial overlap, even if one memory operand is wholly contained within another.

### Flags Affected

None

### Protected Mode Exceptions

#UD if LOCK is used with an instruction not listed in the "Description" section above; other exceptions can be generated by the subsequent (locked) instruction

Real Address Mode Exceptions

Interrupt 6 if LOCK is used with an instruction not listed in the "Description" section above; exceptions can still be generated by the subsequent (locked) instruction

Virtual 8086 Mode Exceptions

#UD if LOCK is used with an instruction not listed in the "Description" section above; exceptions can still be generated by the subsequent (locked) instruction

## LODS/LODSB/LODSW/LODSD —— Load String Operand

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| AC | LODS m8 | 5 | Load byte [(E)SI] into AL |
| AD | LODS m16 | 5 | Load word [(E)SI] into AX |
| AD | LODS m32 | 5 | Load dword [(E)SI] into EAX |
| AC | LODSB | 5 | Load byte DS:[(E)SI] into AL |
| AD | LODSW | 5 | Load word DS:[(E)SI] into AX |
| AD | LODSD | 5 | Load dword DS:[(E)SI] into EAX |

### Operation

```
IF AddressSize = 16
THEN use SI for source-index
ELSE (* AddressSize = 32 *)
   use ESI for source-index;
FI;
IF byte type of instruction
THEN
   AL ← [source-index]; (* byte load *)
   IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← -1; FI;
ELSE
   IF OperandSize = 16
   THEN
      AX ← [source-index]; (* word load *)
      IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← -2; FI;
   ELSE (* OperandSize = 32 *)
      EAX ← [source-index]; (* dword load *)
      IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← -4; FI;
   FI;
FI;
source-index ← source-index + IncDec
```

### Description

LODS loads the AL, AX, or EAX register with the memory byte, word, or doubleword at the location pointed to by the source-index register. After the transfer is made, the source-index register is automatically advanced. If the direction flag is 0 (CLD was executed), the source index increments; if the direction flag is 1 (STD was executed), it decrements. The increment or decrement is 1 if a byte is loaded, 2 if a word is loaded, or 4 if a doubleword is loaded.

If the address-size attribute for this instruction is 16 bits, SI is used for the source-index register; otherwise the address-size attribute is 32 bits, and the ESI register is used. The address of the source data is determined solely by the contents of ESI/SI. Load the correct index value into SI before executing the LODS instruction. LODSB, LODSW, LODSD are synonyms for the byte, word, and doubleword LODS instructions.

LODS can be preceded by the REP prefix; however, LODS is used more typically within a LOOP construct, because further processing of the data moved into EAX, AX, or AL is usually necessary.

Flags Affected

None

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## LOOP/LOOPcond —— Loop Control with CX Counter

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| E2 cb | LOOP rel8 | 11+m | DEC count; jump short if count ≠ 0 |
| E1 cb | LOOPE rel8 | 11+m | DEC count; jump short if count ≠ 0 and ZF=1 |
| E1 cb | LOOPZ rel8 | 11+m | DEC count; jump short if count ≠ 0 and ZF=1 |
| E0 cb | LOOPNE rel8 | 11+m | DEC count; jump short if count ≠ 0 and ZF=0 |
| E0 cb | LOOPNZ rel8 | 11+m | DEC count; jump short if count ≠ 0 and ZF=0 |

### Operation

```
IF AddressSize = 16 THEN CountReg is CX ELSE CountReg is ECX; FI;
CountReg ← CountReg - 1;
IF instruction ≠ LOOP
THEN
   IF (instruction = LOOPE) OR (instruction = LOOPZ)
   THEN BranchCond ← (ZF = 1) AND (CountReg ≠ 0);
   FI;
   IF (instruction = LOOPNE) OR (instruction = LOOPNZ)
   THEN BranchCond ← (ZF = 0) AND (CountReg ≠ 0);
   FI;
FI;

IF BranchCond
THEN
   IF OperandSize = 16
   THEN
      IP ← IP + SignExtend(rel8);
   ELSE (* OperandSize = 32 *)
      EIP ← EIP + SignExtend(rel8);
   FI;
FI;
```

### Description

LOOP decrements the count register without changing any of the flags. Conditions are then checked for the form of LOOP being used. If the conditions are met, a short jump is made to the label given by the operand to LOOP. If the address-size attribute is 16 bits, the CX register is used as the count register; otherwise the ECX register is used. The operand of LOOP must be in the range from 128 (decimal) bytes before the instruction to 127 bytes ahead of the instruction.

The LOOP instructions provide iteration control and combine loop index management with conditional branching. Use the LOOP instruction by loading an unsigned iteration count into the count register, then code the LOOP at the end of a series of instructions to be iterated. The destination of LOOP is a label that points to the beginning of the iteration.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if the offset jumped to is beyond the limits of the current code segment

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

**LSL —— Load Segment Limit**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 03 /r | LSL r16,r/m16 | pm=20/21 | Load: r16 ← segment limit, selector r/m16 (byte granular) |
| 0F 03 /r | LSL r32,r/m32 | pm=20/21 | Load: r32 ← segment limit, selector r/m32 (byte granular) |
| 0F 03 /r | LSL r16,r/m16 | pm=25/26 | Load: r16 ← segment limit, selector r/m16 (page granular) |
| 0F 03 /r | LSL r32,r/m32 | pm=25/26 | Load: r32 ← segment limit, selector r/m32 (page granular) |

Description

The LSL instruction loads a register with an unscrambled segment limit, and sets ZF to 1, provided that the source selector is visible at the CPL weakened by RPL, and that the descriptor is a type accepted by LSL. Otherwise, ZF is cleared to 0, and the destination register is unchanged. The segment limit is loaded as a byte granular value. If the descriptor has a page granular segment limit, LSL will translate it to a byte limit before loading it in the destination register (shift left 12 the 20-bit "raw" limit from descriptor, then OR with 00000FFFH).

The 32-bit forms of this instruction store the 32-bit byte granular limit in the 16-bit destination register.

Code and data segment descriptors are valid for LSL.

The valid special segment and gate descriptor types for LSL are given in the following table:

| Type | Name | Valid/Invalid |
|---|---|---|
| 0 | Invalid | Invalid |
| 1 | Available 80286 TSS | Valid |
| 2 | LDT | Valid |
| 3 | Busy 80286 TSS | Valid |
| 4 | 80286 call gate | Invalid |
| 5 | 80286/80386 task gate | Invalid |
| 6 | 80286 trap gate | Invalid |
| 7 | 80286 interrupt gate | Invalid |
| 8 | Invalid | Valid |
| 9 | Available 80386 TSS | Valid |
| A | Invalid | Invalid |
| B | Busy 80386 TSS | Valid |
| C | 80386 call gate | Invalid |
| D | Invalid | Invalid |
| E | 80386 trap gate | Invalid |
| F | 80386 interrupt gate | Invalid |

Flags Affected

ZF as described above

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 6; LSL is not recognized in Real Address Mode

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode

## LTR —— Load Task Register

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 00 /3 | LTR r/m16 | pm=23/27 | Load EA word into task register |

### Description

LTR loads the task register from the source register or memory location specified by the operand. The loaded task state segment is marked busy. A task switch does not occur.

LTR is used only in operating system software; it is not used in application programs.

### Flags Affected

None

### Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #GP(0) if the current privilege level is not 0; #GP(selector) if the object named by the source selector is not a TSS or is already busy; #NP(selector) if the TSS is marked "not present"; #PF(fault-code) for a page fault

### Real Address Mode Exceptions

Interrupt 6; LTR is not recognized in Real Address Mode

### Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode

### Notes

The operand-size attribute has no effect on this instruction.

**MOV —— Move Data**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 88 /r | MOV r/m8,r8 | 2/2 | Move byte register to r/m byte |
| 89 /r | MOV r/m16,r16 | 2/2 | Move word register to r/m word |
| 89 /r | MOV r/m32,r32 | 2/2 | Move dword register to r/m dword |
| 8A /r | MOV r8,r/m8 | 2/4 | Move r/m byte to byte register |
| 8B /r | MOV r16,r/m16 | 2/4 | Move r/m word to word register |
| 8B /r | MOV r32,r/m32 | 2/4 | Move r/m dword to dword register |
| 8C /r | MOV r/m16,Sreg | 2/2 | Move segment register to r/m word |
| 8D /r | MOV Sreg,r/m16 | 2/5,pm=18/19 | Move r/m word to segment register |
| A0 | MOV AL,moffs8 | 4 | Move byte at (seg:offset) to AL |
| A1 | MOV AX,moffs16 | 4 | Move word at (seg:offset) to AX |
| A1 | MOV EAX,moffs32 | 4 | Move dword at (seg:offset) to EAX |
| A2 | MOV moffs8,AL | 2 | Move AL to (seg:offset) |
| A3 | MOV moffs16,AX | 2 | Move AX to (seg:offset) |
| A3 | MOV moffs32,EAX | 2 | Move EAX to (seg:offset) |
| B0 + rb | MOV reg8,imm8 | 2 | Move immediate byte to register |
| B8 + rw | MOV reg16,imm16 | 2 | Move immediate word to register |
| B8 + rd | MOV reg32,imm32 | 2 | Move immediate dword to register |
| Ciiiiii | MOV r/m8,imm8 | 2/2 | Move immediate byte to r/m byte |
| C7 | MOV r/m16,imm16 | 2/2 | Move immediate word to r/m word |
| C7 | MOV r/m32,imm32 | 2/2 | Move immediate dword to r/m dword |

---

NOTES:
moffs8, moffs16, and moffs32 all consist of a simple offset relative to the segment base. The 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.

---

Operation

```
DEST ← SRC;
```

Description

MOV copies the second operand to the first operand.

If the destination operand is a segment register (DS, ES, SS, etc.), then data from a descriptor is also loaded into the register. The data fo register is obtained from the descriptor table entry for the selector given. A null selector (values 0000-0003) can be loaded into DS and ES registers without causing an exception; however, use of DS or ES causes a #GP(0), and no memory reference occurs.

A MOV into SS inhibits all interrupts until after the execution of the next instruction (which is presumably a MOV into eSP).

Loading a segment register under 80386 Protected Mode results in special checks and actions, as described in the following listing:

```
IF SS is loaded;
THEN
   IF selector is null THEN #GP(0);
FI;
```

```
   Selector index must be within its descriptor table limits else
      #GP(selector);
   Selector's RPL must equal CPL else #GP(selector);
AR byte must indicate a writable data segment else #GP(selector);
   DPL in the AR byte must equal CPL else #GP(selector);
   Segment must be marked present else #SS(selector);
   Load SS with selector;
   Load SS with descriptor.
FI;
IF DS, ES, FS or GS is loaded with non-null selector;
THEN
   Selector index must be within its descriptor table limits
      else #GP(selector);
   AR byte must indicate data or readable code segment else
      #GP(selector);
   IF data or nonconforming code segment
   THEN both the RPL and the CPL must be less than or equal to DPL in
      AR byte;
   ELSE #GP(selector);
   FI;
   Segment must be marked present else #NP(selector);
   Load segment register with selector;
   Load segment register with descriptor;
FI;
IF DS, ES, FS or GS is loaded with a null selector;
THEN
   Load segment register with selector;
   Clear descriptor valid bit;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP, #SS, and #NP if a segment register is being loaded; otherwise, #GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**MOV —— Move to/from Special Registers**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 20 /r | MOV r32,CR0/CR2/CR3 | 6 | Move (control register) to (register) |
| 0F 22 /r | MOV CR0/CR2/CR3,r32 | 10/4/5 | Move (register) to (control register) |
| 0F 21 /r | MOV r32,DR0 -- 3 | 22 | Move (debug register) to (register) |
| 0F 21 /r | MOV r32,DR6/DR7 | 14 | Move (debug register) to (register) |
| 0F 23 /r | MOV DR0 -- 3,r32 | 22 | Move (register) to (debug register) |
| 0F 23 /r | MOV DR6/DR7,r32 | 16 | Move (register) to (debug register) |
| 0F 24 /r | MOV r32,TR6/TR7 | 12 | Move (test register) to (register) |
| 0F 26 /r | MOV TR6/TR7,r32 | 12 | Move (register) to (test register) |

Operation

DEST ← SRC;

Description

The above forms of MOV store or load the following special registers in or from a general purpose register:

- Control registers CR0, CR2, and CR3
- Debug Registers DR0, DR1, DR2, DR3, DR6, and DR7
- Test Registers TR6 and TR7

32-bit operands are always used with these instructions, regardless of the operand-size attribute.

Flags Affected

OF, SF, ZF, AF, PF, and CF are undefined

Protected Mode Exceptions

#GP(0) if the current privilege level is not 0

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

#GP(0) if instruction execution is attempted

Notes

The instructions must be executed at privilege level 0 or in real-address mode; otherwise, a protection exception will be raised.

The reg field within the ModRM byte specifies which of the special registers in each category is involved. The two bits in the  field are always 11. The r/m field specifies the general register involved.

**MOVS/MOVSB/MOVSW/MOVSD —— Move Data from String to String**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| A4 | MOVS m8,m8 | 7 | Move byte [(E)SI] to ES:[(E)DI] |
| A5 | MOVS m16,m16 | 7 | Move word [(E)SI] to ES:[(E)DI] |
| A5 | MOVS m32,m32 | 7 | Move dword [(E)SI] to ES:[(E)DI] |
| A4 | MOVSB | 7 | Move byte DS:[(E)SI] to ES:[(E)DI] |
| A5 | MOVSW | 7 | Move word DS:[(E)SI] to ES:[(E)DI] |
| A5 | MOVSD | 7 | Move dword DS:[(E)SI] to ES:[(E)DI] |

Operation

```
IF (instruction = MOVSD) OR (instruction has doubleword operands)
THEN OperandSize ← 32;
ELSE OperandSize ← 16;
IF AddressSize = 16
THEN use SI for source-index and DI for destination-index;
ELSE (* AddressSize = 32 *)
   use ESI for source-index and EDI for destination-index;
FI;
IF byte type of instruction
THEN
   [destination-index] ← [source-index]; (* byte assignment *)
   IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← -1; FI;
ELSE
   IF OperandSize = 16
   THEN
      [destination-index] ← [source-index]; (* word assignment *)
      IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← -2; FI;
   ELSE (* OperandSize = 32 *)
      [destination-index] ← [source-index]; (* doubleword assignment *)
      IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← -4; FI;
   FI;
FI;
source-index ← source-index + IncDec;
destination-index ← destination-index + IncDec;
```

Description

MOVS copies the byte or word at [(E)SI] to the byte or word at ES:[(E)DI]. The destination operand must be addressable from the ES register; no segment override is possible for the destination. A segment override can be used for the source operand; the default is DS.

The addresses of the source and destination are determined solely by the contents of (E)SI and (E)DI. Load the correct index values into (E)SI and (E)DI before executing the MOVS instruction. MOVSB, MOVSW, and MOVSD are synonyms for the byte, word, and doubleword MOVS instructions.

After the data is moved, both (E)SI and (E)DI are advanced automatically. If the direction flag is 0 (CLD was executed), the registers are incremented; if the direction flag is 1 (STD was executed), the registers are decremented. The registers are incremented or decremented by 1 if a byte was moved, 2 if a word was moved, or 4 if a doubleword was moved.

MOVS can be preceded by the REP prefix for block movement of CX bytes or words. Refer to the REP instruction for details of this operation.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**MOVSX —— Move with Sign-Extend**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F BE /r | MOVSX r16,r/m8 | 3/6 | Move byte to word with sign-extend |
| 0F BE /r | MOVSX r32,r/m8 | 3/6 | Move byte to dword, sign-extend |
| 0F BF /r | MOVSX r32,r/m16 | 3/6 | Move word to dword, sign-extend |

Operation

DEST ← SignExtend(SRC);

Description

MOVSX reads the contents of the effective address or register as a byte or a word, sign-extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

Flags Affected

None

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**MOVZX —— Move with Zero-Extend**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F B6 /r | MOVZX r16,r/m8 | 3/6 | Move byte to word with zero-extend |
| 0F B6 /r | MOVZX r32,r/m8 | 3/6 | Move byte to dword, zero-extend |
| 0F B7 /r | MOVZX r32,r/m16 | 3/6 | Move word to dword, zero-extend |

Operation

DEST ← ZeroExtend(SRC);

Description

MOVZX reads the contents of the effective address or register as a byte or a word, zero extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

Flags Affected

None

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**MUL —— Unsigned Multiplication of AL or AX**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F6 /4 | MUL AL,r/m8 | 9-14/12-17 | Unsigned multiply (AX ← AL * r/m byte) |
| F7 /4 | MUL AX,r/m16 | 9-22/12-25 | Unsigned multiply (DX:AX ← AX * r/m word) |
| F7 /4 | MUL EAX,r/m32 | 9-38/12-41 | Unsigned multiply (EDX:EAX ← EAX * r/m dword) |

---

NOTES:
The 80386 uses an early-out multiply algorithm. The actual number of clocks depends on the position of the most significant bit in the optimizing multiplier, shown underlined above. The optimization occurs for positive and negative multiplier values. Because of the early-out algorithm, clock counts given are minimum to maximum. To calculate the actual clocks, use the following formula:

Actual clock = if ≠ 0 then max(ceiling($\log_2 |m|$), 3) + 6 clocks;

Actual clock = if = 0 then 9 clocks

where m is the multiplier.

---

Operation

```
IF byte-size operation
THEN AX ← AL * r/m8
ELSE (* word or doubleword operation *)
   IF OperandSize = 16
   THEN DX:AX ← AX * r/m16
   ELSE (* OperandSize = 32 *)
      EDX:EAX ← EAX * r/m32
   FI;
FI;
```

Description

MUL performs unsigned multiplication. Its actions depend on the size of its operand, as follows:

- A byte operand is multiplied by AL; the result is left in AX. The carry and overflow flags are set to 0 if AH is 0; otherwise, they are set to 1.
- A word operand is multiplied by AX; the result is left in DX:AX. DX contains the high-order 16 bits of the product. The carry and overflow flags are set to 0 if DX is 0; otherwise, they are set to 1.
- A doubleword operand is multiplied by EAX and the result is left in EDX:EAX. EDX contains the high-order 32 bits of the product. The carry and overflow flags are set to 0 if EDX is 0; otherwise, they are set to 1.

Flags Affected

OF and CF as described above; SF, ZF, AF, PF, and CF are undefined

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## NEG —— Two's Complement Negation

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F6 /3 | NEG r/m8 | 2/6 | Two's complement negate r/m byte |
| F7 /3 | NEG r/m16 | 2/6 | Two's complement negate r/m word |
| F7 /3 | NEG r/m32 | 2/6 | Two's complement negate r/m dword |

### Operation

```
IF r/m = 0 THEN CF ← 0 ELSE CF ← 1; FI;
r/m ← - r/m;
```

### Description

NEG replaces the value of a register or memory operand with its two's complement. The operand is subtracted from zero, and the result is placed in the operand.

The carry flag is set to 1, unless the operand is zero, in which case the carry flag is cleared to 0.

### Flags Affected

CF as described above; OF, SF, ZF, and PF as described in Appendix C

### Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

### Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

### Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault

**NOP —— No Operation**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 90 | NOP | 3 | No operation |

Description

NOP performs no operation. NOP is a one-byte instruction that takes up space but affects none of the machine context except (E)IP.

NOP is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

Flags Affected

None

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

**NOT —— One's Complement Negation**

| Opcode | | Instruction | Clocks | Description |
|---|---|---|---|---|
| F6 | /2 | NOT r/m8 | 2/6 | Reverse each bit of r/m byte |
| F7 | /2 | NOT r/m16 | 2/6 | Reverse each bit of r/m word |
| F7 | /2 | NOT r/m32 | 2/6 | Reverse each bit of r/m dword |

Operation

r/m ← NOT r/m;

Description

NOT inverts the operand; every 1 becomes a 0, and vice versa.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault

**OR —— Logical Inclusive OR**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0C ib | OR AL,imm8 | 2 | OR immediate byte to AL |
| 0D iw | OR AX,imm16 | 2 | OR immediate word to AX |
| 0D id | OR EAX,imm32 | 2 | OR immediate dword to EAX |
| 80 /1 ib | OR r/m8,imm8 | 2/7 | OR immediate byte to r/m byte |
| 81 /1 iw | OR r/m16,imm16 | 2/7 | OR immediate word to r/m word |
| 81 /1 id | OR r/m32,imm32 | 2/7 | OR immediate dword to r/m dword |
| 83 /1 ib | OR r/m16,imm8 | 2/7 | OR sign-extended immediate byte with r/m word |
| 83 /1 ib | OR r/m32,imm8 | 2/7 | OR sign-extended immediate byte with r/m dword |
| 08 /r | OR r/m8,r8 | 2/6 | OR byte register to r/m byte |
| 09 /r | OR r/m16,r16 | 2/6 | OR word register to r/m word |
| 09 /r | OR r/m32,r32 | 2/6 | OR dword register to r/m dword |
| 0A /r | OR r8,r/m8 | 2/7 | OR byte register to r/m byte |
| 0B /r | OR r16,r/m16 | 2/7 | OR word register to r/m word |
| 0B /r | OR r32,r/m32 | 2/7 | OR dword register to r/m dword |

Operation

```
DEST ← DEST OR SRC;
CF ← 0;
OF ← 0
```

Description

OR computes the inclusive OR of its two operands and places the result in the first operand. Each bit of the result is 0 if both corresponding bits of the operands are 0; otherwise, each bit is 1.

Flags Affected

OF ← 0, CF ← 0; SF, ZF, and PF as described in Appendix C; AF is undefined

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page Fault.

**OUT —— Output to Port**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| E6 ib | OUT imm8,AL | 10,pm=4*/24** | Output byte AL to immediate port number |
| E7 ib | OUT imm8,AX | 10,pm=4*/24** | Output word AL to immediate port number |
| E7 ib | OUT imm8,EAX | 10,pm=4*/24** | Output dword AL to immediate port number |
| EE | OUT DX,AL | 11,pm=5*/25** | Output byte AL to port number in DX |
| EF | OUT DX,AX | 11,pm=5*/25** | Output word AL to port number in DX |
| EF | OUT DX,EAX | 11,pm=5*/25** | Output dword AL to port number in DX |

NOTES:
*If CPL ≤ IOPL
**If CPL > IOPL or if in virtual 8086 mode

Operation

```
IF (PE = 1) AND ((VM = 1) OR (CPL > IOPL))
THEN (* Virtual 8086 mode, or protected mode with CPL > IOPL *)
   IF NOT I-O-Permission (DEST, width(DEST))
   THEN #GP(0);
   FI;
FI;
[DEST] ← SRC; (* I/O address space used *)
```

Description

OUT transfers a data byte or data word from the register (AL, AX, or EAX) given as the second operand to the output port numbered by the first operand. Output to any port from 0 to 65535 is performed by placing the port number in the DX register and then using an OUT instruction with DX as the first operand. If the instruction contains an eight-bit port ID, that value is zero-extended to 16 bits.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if the current privilege level is higher (has less privilege) than IOPL and any of the corresponding I/O permission bits in TSS equals 1

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

#GP(0) fault if any of the corresponding I/O permission bits in TSS equals 1

## OUTS/OUTSB/OUTSW/OUTSD —— Output String to Port

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 6E | OUTS DX,r/m8 | 14,pm=8*/28** | Output byte [(E)SI] to port in DX |
| 6F | OUTS DX,r/m16 | 14,pm=8*/28** | Output word [(E)SI] to port in DX |
| 6F | OUTS DX,r/m32 | 14,pm=8*/28** | Output dword [(E)SI] to port in DX |
| 6E | OUTSB | 14,pm=8*/28** | Output byte DS:[(E)SI] to port in DX |
| 6F | OUTSW | 14,pm=8*/28** | Output word DS:[(E)SI] to port in DX |
| 6F | OUTSD | 14,pm=8*/28** | Output dword DS:[(E)SI] to port in DX |

---

NOTES:
*If CPL ≤ IOPL
**If CPL > IOPL or if in virtual 8086 mode

---

Operation

```
IF AddressSize = 16
THEN use SI for source-index;
ELSE (* AddressSize = 32 *)
   use ESI for source-index;
FI;

IF (PE = 1) AND ((VM = 1) OR (CPL > IOPL))
THEN (* Virtual 8086 mode, or protected mode with CPL > IOPL *)
   IF NOT I-O-Permission (DEST, width(DEST))
   THEN #GP(0);
   FI;
FI;
IF byte type of instruction
THEN
   [DX] ← [source-index]; (* Write byte at DX I/O address *)
   IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← -1; FI;
FI;
IF OperandSize = 16
THEN
   [DX] ← [source-index]; (* Write word at DX I/O address *)
   IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← -2; FI;
FI;
IF OperandSize = 32
THEN
   [DX] ← [source-index]; (* Write dword at DX I/O address *)
   IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← -4; FI;
   FI;
FI;
source-index ← source-index + IncDec;
```

Description

OUTS transfers data from the memory byte, word, or doubleword at the source-index register to the output port addressed by the DX register. If the address-size attribute for this instruction is 16 bits, SI is used for the source-index register; otherwise, the address-size attribute is 32 bits, and ESI is used for the source-index register.

OUTS does not allow specification of the port number as an immediate value. The port must be addressed through the DX register value. Load the correct value into DX before executing the OUTS instruction.

The address of the source data is determined by the contents of source-index register. Load the correct index value into SI or ESI before executing the OUTS instruction.

After the transfer, source-index register is advanced automatically. If the direction flag is 0 (CLD was executed), the source-index register is incremented; if the direction flag is 1 (STD was executed), it is decremented. The amount of the increment or decrement is 1 if a byte is output, 2 if a word is output, or 4 if a doubleword is output.

OUTSB, OUTSW, and OUTSD are synonyms for the byte, word, and doubleword OUTS instructions. OUTS can be preceded by the REP prefix for block output of CX bytes or words. Refer to the REP instruction for details on this operation.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if CPL is greater than IOPL and any of the corresponding I/O permission bits in TSS equals 1; #GP(0) for an illegal memory operand effective address in the CS, DS, or ES segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

#GP(0) fault if any of the corresponding I/O permission bits in TSS equals 1; #PF(fault-code) for a page fault

**POP —— Pop a Word from the Stack**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 8F /0 | POP m16 | 5 | Pop top of stack into memory word |
| 8F /0 | POP m32 | 5 | Pop top of stack into memory dword |
| 58 + rw | POP r16 | 4 | Pop top of stack into word register |
| 58 + rd | POP r32 | 4 | Pop top of stack into dword register |
| 1F | POP DS | 7,pm=21 | Pop top of stack into DS |
| 07 | POP ES | 7,pm=21 | Pop top of stack into ES |
| 17 | POP SS | 7,pm=21 | Pop top of stack into SS |
| 0F A1 | POP FS | 7,pm=21 | Pop top of stack into FS |
| 0F A9 | POP GS | 7,pm=21 | Pop top of stack into GS |

Operation

```
IF StackAddrSize = 16
THEN
   IF OperandSize = 16
   THEN
      DEST ← (SS:SP); (* copy a word *)
      SP ← SP + 2;
   ELSE (* OperandSize = 32 *)
      DEST ← (SS:SP); (* copy a dword *)
      SP ← SP + 4;
   FI;
ELSE (* StackAddrSize = 32 * )
   IF OperandSize = 16
   THEN
      DEST ← (SS:ESP); (* copy a word *)
      ESP ← ESP + 2;
   ELSE (* OperandSize = 32 *)
      DEST ← (SS:ESP); (* copy a dword *)
      ESP ← ESP + 4;
   FI;
FI;
```

Description

POP replaces the previous contents of the memory, the register, or the segment register operand with the word on the top of the 80386 stack, addressed by SS:SP (address-size attribute of 16 bits) or SS:ESP (addresssize attribute of 32 bits). The stack pointer SP is incremented by 2 for an operand-size of 16 bits or by 4 for an operand-size of 32 bits. It then points to the new top of stack.

POP CS is not an 80386 instruction. Popping from the stack into the CS register is accomplished with a RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the value popped must be a selector. In protected mode, loading the selector initiates automatic loading of the descriptor information associated with that selector into the hidden part of the segment register; loading also initiates validation of both the selector and the descriptor information.

A null value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a protection exception. An attempt to reference a segment whose corresponding segment register is loaded with a null value causes a #GP(0) exception. No memory reference occurs. The saved value of the segment register is null.

A POP SS instruction inhibits all interrupts, including NMI, until after execution of the next instruction. This allows sequential execution of POP SS and POP eSP instructions without danger of having an invalid stack during an interrupt. However, use of the LSS instruction is the preferred method of loading the SS and eSP registers.

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing:

```
IF SS is loaded:
   IF selector is null THEN #GP(0);
   Selector index must be within its descriptor table limits ELSE
      #GP(selector);
   Selector's RPL must equal CPL ELSE #GP(selector);
   AR byte must indicate a writable data segment ELSE #GP(selector);
   DPL in the AR byte must equal CPL ELSE #GP(selector);
   Segment must be marked present ELSE #SS(selector);
   Load SS register with selector;
   Load SS register with descriptor;

IF DS, ES, FS or GS is loaded with non-null selector:
   AR byte must indicate data or readable code segment ELSE
      #GP(selector);
   IF data or nonconforming code
   THEN both the RPL and the CPL must be less than or equal to DPL in
      AR byte
   ELSE #GP(selector);
   FI;
   Segment must be marked present ELSE #NP(selector);
   Load segment register with selector;
   Load segment register with descriptor;

IF DS, ES, FS, or GS is loaded with a null selector:
   Load segment register with selector
   Clear valid bit in invisible portion of register
```

Flags Affected

None

Protected Mode Exceptions

#GP, #SS, and #NP if a segment register is being loaded; #SS(0) if the current top of stack is not within the stack segment; #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault

## POPA/POPAD —— Pop all General Registers

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 61 | POPA | 24 | Pop DI, SI, BP, SP, BX, DX, CX, and AX |
| 61 | POPAD | 24 | Pop EDI, ESI, EBP, ESP, EDX, ECX, and EAX |

### Operation

```
IF OperandSize = 16 (* instruction = POPA *)
THEN
   DI ← Pop();
   SI ← Pop();
   BP ← Pop();
   throwaway ← Pop (); (* Skip SP *)
   BX ← Pop();
   DX ← Pop();
   CX ← Pop();
   AX ← Pop();
ELSE (* OperandSize = 32, instruction = POPAD *)
   EDI ← Pop();
   ESI ← Pop();
   EBP ← Pop();
   throwaway ← Pop (); (* Skip ESP *)
   EBX ← Pop();
   EDX ← Pop();
   ECX ← Pop();
   EAX ← Pop();
FI;
```

### Description

POPA pops the eight 16-bit general registers. However, the SP value is discarded instead of loaded into SP. POPA reverses a previous PUSHA, restoring the general registers to their values before PUSHA was executed. The first register popped is DI.

POPAD pops the eight 32-bit general registers. The ESP value is discarded instead of loaded into ESP. POPAD reverses the previous PUSHAD, restoring the general registers to their values before PUSHAD was executed. The first register popped is EDI.

### Flags Affected

None

### Protected Mode Exceptions

#SS(0) if the starting or ending stack address is not within the stack segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault

## POPF/POPFD —— Pop Stack into FLAGS or EFLAGS Register

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 9D | POPF | 5 | Pop top of stack FLAGS |
| 9D | POPFD | 5 | Pop top of stack into EFLAGS |

### Operation

```
Flags ← Pop();
```

### Description

POPF/POPFD pops the word or doubleword on the top of the stack and stores the value in the flags register. If the operand-size attribute of the instruction is 16 bits, then a word is popped and the value is stored in FLAGS. If the operand-size attribute is 32 bits, then a doubleword is popped and the value is stored in EFLAGS.

Refer to Chapter 2 and Chapter 4 for information about the FLAGS and EFLAGS registers. Note that bits 16 and 17 of EFLAGS, called VM and RF, respectively, are not affected by POPF or POPFD.

The I/O privilege level is altered only when executing at privilege level 0. The interrupt flag is altered only when executing at a level at least as privileged as the I/O privilege level. (Real-address mode is equivalent to privilege level 0.) If a POPF instruction is executed with insufficient privilege, an exception does not occur, but the privileged bits do not change.

### Flags Affected

All flags except VM and RF

### Protected Mode Exceptions

#SS(0) if the top of stack is not within the stack segment

### Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

### Virtual 8086 Mode Exceptions

#GP(0) fault if IOPL is less than 3, to permit emulation

**PUSH —— Push Operand onto the Stack**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| FF /6 | PUSH m16 | 5 | Push memory word |
| FF /6 | PUSH m32 | 5 | Push memory dword |
| 50 + /r | PUSH r16 | 2 | Push register word |
| 50 + /r | PUSH r32 | 2 | Push register dword |
| 6A | PUSH imm8 | 2 | Push immediate byte |
| 68 | PUSH imm16 | 2 | Push immediate word |
| 68 | PUSH imm32 | 2 | Push immediate dword |
| 0E | PUSH CS | 2 | Push CS |
| 16 | PUSH SS | 2 | Push SS |
| 1E | PUSH DS | 2 | Push DS |
| 06 | PUSH ES | 2 | Push ES |
| 0F A0 | PUSH FS | 2 | Push FS |
| OF A8 | PUSH GS | 2 | Push GS |

Operation

```
IF StackAddrSize = 16
THEN
   IF OperandSize = 16 THEN
      SP ← SP - 2;
      (SS:SP) ← (SOURCE); (* word assignment *)
   ELSE
      SP ← SP - 4;
      (SS:SP) ← (SOURCE); (* dword assignment *)
   FI;
ELSE (* StackAddrSize = 32 *)
   IF OperandSize = 16
   THEN
      ESP ← ESP - 2;
      (SS:ESP) ← (SOURCE); (* word assignment *)
   ELSE
      ESP ← ESP - 4;
      (SS:ESP) ← (SOURCE); (* dword assignment *)
   FI;
FI;
```

Description

PUSH decrements the stack pointer by 2 if the operand-size attribute of the instruction is 16 bits; otherwise, it decrements the stack pointer by 4. PUSH then places the operand on the new top of stack, which is pointed to by the stack pointer.

The 80386 PUSH eSP instruction pushes the value of eSP as it existed before the instruction. This differs from the 8086, where PUSH SP pushes the new value (decremented by 2).

Flags Affected

None

Protected Mode Exceptions

#SS(0) if the new value of SP or ESP is outside the stack segment limit; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

None; if SP or ESP is 1, the 80386 shuts down due to a lack of stack space

Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault

## PUSHA/PUSHAD —— Push all General Registers

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 60 | PUSHA | 18 | Push AX, CX, DX, BX, original SP, BP, SI, and DI |
| 60 | PUSHAD | 18 | Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI |

### Operation

```
IF OperandSize = 16 (* PUSHA instruction *)
THEN
   Temp ← (SP);
   Push(AX);
   Push(CX);
   Push(DX);
   Push(BX);
   Push(Temp);
   Push(BP);
   Push(SI);
   Push(DI);
ELSE (* OperandSize = 32, PUSHAD instruction *)
   Temp ← (ESP);
   Push(EAX);
   Push(ECX);
   Push(EDX);
   Push(EBX);
   Push(Temp);
   Push(EBP);
   Push(ESI);
   Push(EDI);
FI;
```

### Description

PUSHA and PUSHAD save the 16-bit or 32-bit general registers, respectively, on the 80386 stack. PUSHA decrements the stack pointer (SP) by 16 to hold the eight word values. PUSHAD decrements the stack pointer (ESP) by 32 to hold the eight doubleword values. Because the registers are pushed onto the stack in the order in which they were given, they appear in the 16 or 32 new stack bytes in reverse order. The last register pushed is DI or EDI.

### Flags Affected

None

### Protected Mode Exceptions

#SS(0) if the starting or ending stack address is outside the stack segment limit; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Before executing PUSHA or PUSHAD, the 80386 shuts down if SP or ESP equals 1, 3, or 5; if SP or ESP equals 7, 9, 11, 13, or 15, exception 13 occurs

Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault

**PUSHF/PUSHFD —— Push Flags Register onto the Stack**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 9C | PUSHF | 4 | Push FLAGS |
| 9C | PUSHFD | 4 | Push EFLAGS |

Operation

```
IF OperandSize = 32
THEN push(EFLAGS);
ELSE push(FLAGS);
FI;
```

Description

PUSHF decrements the stack pointer by 2 and copies the FLAGS register to the new top of stack; PUSHFD decrements the stack pointer by 4, and the 80386 EFLAGS register is copied to the new top of stack which is pointed to by SS:eSP. Refer to Chapter 2 and Chapter 4 for information on the EFLAGS register.

Flags Affected

None

Protected Mode Exceptions

#SS(0) if the new value of eSP is outside the stack segment boundaries

Real Address Mode Exceptions

None; the 80386 shuts down due to a lack of stack space

Virtual 8086 Mode Exceptions

#GP(0) fault if IOPL is less than 3, to permit emulation

## RCL/RCR/ROL/ROR —— Rotate

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| D0 /2 | RCL r/m8,1 | 9/10 | Rotate 9 bits (CF,r/m byte) left once |
| D2 /2 | RCL r/m8,CL | 9/10 | Rotate 9 bits (CF,r/m byte) left CL times |
| C0 /2 ib | RCL r/m8,imm8 | 9/10 | Rotate 9 bits (CF,r/m byte) left imm8 times |
| D1 /2 | RCL r/m16,1 | 9/10 | Rotate 17 bits (CF,r/m word) left once |
| D3 /2 | RCL r/m16,CL | 9/10 | Rotate 17 bits (CF,r/m word) left CL times |
| C1 /2 ib | RCL r/m16,imm8 | 9/10 | Rotate 17 bits (CF,r/m word) left imm8 times |
| D1 /2 | RCL r/m32,1 | 9/10 | Rotate 33 bits (CF,r/m dword) left once |
| D3 /2 | RCL r/m32,CL | 9/10 | Rotate 33 bits (CF,r/m dword) left CL times |
| C1 /2 ib | RCL r/m32,imm8 | 9/10 | Rotate 33 bits (CF,r/m dword) left imm8 times |
| D0 /3 | RCR r/m8,1 | 9/10 | Rotate 9 bits (CF,r/m byte) right once |
| D2 /3 | RCR r/m8,CL | 9/10 | Rotate 9 bits (CF,r/m byte) right CL times |
| C0 /3 ib | RCR r/m8,imm8 | 9/10 | Rotate 9 bits (CF,r/m byte) right imm8 times |
| D1 /3 | RCR r/m16,1 | 9/10 | Rotate 17 bits (CF,r/m word) right once |
| D3 /3 | RCR r/m16,CL | 9/10 | Rotate 17 bits (CF,r/m word) right CL times |
| C1 /3 ib | RCR r/m16,imm8 | 9/10 | Rotate 17 bits (CF,r/m word) right imm8 times |
| D1 /3 | RCR r/m32,1 | 9/10 | Rotate 33 bits (CF,r/m dword) right once |
| D3 /3 | RCR r/m32,CL | 9/10 | Rotate 33 bits (CF,r/m dword) right CL times |
| C1 /3 ib | RCR r/m32,imm8 | 9/10 | Rotate 33 bits (CF,r/m dword) right imm8 times |
| D0 /0 | ROL r/m8,1 | 3/7 | Rotate 8 bits r/m byte left once |
| D2 /0 | ROL r/m8,CL | 3/7 | Rotate 8 bits r/m byte left CL times |
| C0 /0 ib | ROL r/m8,imm8 | 3/7 | Rotate 8 bits r/m byte left imm8 times |
| D1 /0 | ROL r/m16,1 | 3/7 | Rotate 16 bits r/m word left once |
| D3 /0 | ROL r/m16,CL | 3/7 | Rotate 16 bits r/m word left CL times |
| C1 /0 ib | ROL r/m16,imm8 | 3/7 | Rotate 16 bits r/m word left imm8 times |
| D1 /0 | ROL r/m32,1 | 3/7 | Rotate 32 bits r/m dword left once |
| D3 /0 | ROL r/m32,CL | 3/7 | Rotate 32 bits r/m dword left CL times |
| C1 /0 ib | ROL r/m32,imm8 | 3/7 | Rotate 32 bits r/m dword left imm8 times |
| D0 /1 | ROR r/m8,1 | 3/7 | Rotate 8 bits r/m byte right once |
| D2 /1 | ROR r/m8,CL | 3/7 | Rotate 8 bits r/m byte right CL times |
| C0 /1 ib | ROR r/m8,imm8 | 3/7 | Rotate 8 bits r/m word right imm8 times |
| D1 /1 | ROR r/m16,1 | 3/7 | Rotate 16 bits r/m word right once |
| D3 /1 | ROR r/m16,CL | 3/7 | Rotate 16 bits r/m word right CL times |
| C1 /1 ib | ROR r/m16,imm8 | 3/7 | Rotate 16 bits r/m word right imm8 times |
| D1 /1 | ROR r/m32,1 | 3/7 | Rotate 32 bits r/m dword right once |
| D3 /1 | ROR r/m32,CL | 3/7 | Rotate 32 bits r/m dword right CL times |
| C1 /1 ib | ROR r/m32,imm8 | 3/7 | Rotate 32 bits r/m dword right imm8 times |

Operation

```
(* ROL - Rotate Left *)
temp ← COUNT;
WHILE (temp ≠ 0)
DO
   tmpcf ← high-order bit of (r/m);
   r/m ← r/m * 2 + (tmpcf);
   temp ← temp - 1;
OD;
IF COUNT = 1
THEN
   IF high-order bit of r/m ≠ CF
   THEN OF ← 1;
   ELSE OF ← 0;
   FI;
ELSE OF ← undefined;
FI;
(* ROR - Rotate Right *)
temp ← COUNT;
```

```
WHILE (temp ≠ 0 )
DO
   tmpcf ← low-order bit of (r/m);
   r/m ← r/m / 2 + (tmpcf * 2^(width(r/m)));
   temp ← temp - 1;
DO;
IF COUNT = 1
THEN
   IF (high-order bit of r/m) ≠ (bit next to high-order bit of r/m)
   THEN OF ← 1;
   ELSE OF ← 0;
   FI;
ELSE OF ← undefined;
FI;
```

Description

Each rotate instruction shifts the bits of the register or memory operand given. The left rotate instructions shift all the bits upward, except for the top bit, which is returned to the bottom. The right rotate instructions do the reverse: the bits shift downward until the bottom bit arrives at the top.

For the RCL and RCR instructions, the carry flag is part of the rotated quantity. RCL shifts the carry flag into the bottom bit and shifts the top bit into the carry flag; RCR shifts the carry flag into the top bit and shifts the bottom bit into the carry flag. For the ROL and ROR instructions, the original value of the carry flag is not a part of the result, but the carry flag receives a copy of the bit that was shifted from one end to the other.

The rotate is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum instruction execution time, the 80386 does not allow rotation counts greater than 31. If a rotation count greater than 31 is attempted, only the bottom five bits of the rotation are used. The 8086 does not mask rotation counts. The 80386 in Virtual 8086 Mode does mask rotation counts.

The overflow flag is defined only for the single-rotate forms of the instructions (second operand = 1). It is undefined in all other cases. For left shifts/rotates, the CF bit after the shift is XORed with the high-order result bit. For right shifts/rotates, the high-order two bits of the result are XORed to get OF.

Flags Affected

OF only for single rotates; OF is undefined for multi-bit rotates; CF as described above

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## REP/REPE/REPZ/REPNE/REPNZ —— Repeat Following String Operation

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F3 6C | REP INS r/m8, DX | 13+6*(E)CX, | |
| | | pm=7+6*(E)CX | |
| If CPL ≤ IOPL/ | | 27+6*(E)CX | |
| If CPL > IOPL or if in virtual 8086 mode | | | Input (E)CX bytes from port DX into ES:[(E)DI] |
| F3 6D | REP INS r/m16,DX | 13+6*(E)CX, | |
| | | pm=7+6*(E)CX | |
| If CPL ≤ IOPL/ | | 27+6*(E)CX | |
| If CPL > IOPL or if in virtual 8086 mode | | | Input (E)CX words from port DX into ES:[(E)DI] |
| F3 6D | REP INS r/m32,DX | 13+6*(E)CX, | |
| | | pm=7+6*(E)CX | |
| If CPL ≤ IOPL/ | | 27+6*(E)CX | |
| If CPL > IOPL or if in virtual 8086 mode | | | Input (E)CX dwords from port DX into ES:[(E)DI] |
| F3 A4 | REP MOVS m8,m8 | 5+4*(E)CX | Move (E)CX bytes from [(E)SI] to ES:[(E)DI] |
| F3 A5 | REP MOVS m16,m16 | 5+4*(E)CX | Move (E)CX words from [(E)SI] to ES:[(E)DI] |
| F3 A5 | REP MOVS m32,m32 | 5+4*(E)CX | Move (E)CX dwords from [(E)SI] to ES:[(E)DI] |
| F3 6E | REP OUTS DX,r/m8 | 5+12*(E)CX, | |
| | | pm=6+5*(E)CX | |
| If CPL ≤ IOPL/ | | 26+5*(E)CX | |
| If CPL > IOPL or if in virtual 8086 mode | | | Output (E)CX bytes from [(E)SI] to port DX |
| F3 6F | REP OUTS DX,r/m16 | 5+12*(E)CX, | |
| | | pm=6+5*(E)CX | |
| If CPL ≤ IOPL/ | | 26+5*(E)CX | |
| If CPL > IOPL or if in virtual 8086 mode | | | Output (E)CX words from [(E)SI] to port DX |
| F3 6F | REP OUTS DX,r/m32 | 5+12*(E)CX, | |
| | | pm=6+5*(E)CX | |
| If CPL ≤ IOPL/ | | 26+5*(E)CX | |
| If CPL > IOPL or if in virtual 8086 mode | | | Output (E)CX dwords from [(E)SI] to port DX |
| F3 AA | REP STOS m8 | 5+5*(E)CX | Fill (E)CX bytes at ES:[(E)DI] with AL |
| F3 AB | REP STOS m16 | 5+5*(E)CX | Fill (E)CX words at ES:[(E)DI] with AX |
| F3 AB | REP STOS m32 | 5+5*(E)CX | Fill (E)CX dwords at ES:[(E)DI] with EAX |
| F3 A6 | REPE CMPS m8,m8 | 5+9*N | Find nonmatching bytes in ES:[(E)DI] and [(E)SI] |
| F3 A7 | REPE CMPS m16,m16 | 5+9*N | Find nonmatching words in ES:[(E)DI] and [(E)SI] |
| F3 A7 | REPE CMPS m32,m32 | 5+9*N | Find nonmatching dwords in ES:[(E)DI] and [(E)SI] |
| F3 AE | REPE SCAS m8 | 5+8*N | Find non-AL byte starting at ES:[(E)DI] |
| F3 AF | REPE SCAS m16 | 5+8*N | Find non-AX word starting at ES:[(E)DI] |
| F3 AF | REPE SCAS m32 | 5+8*N | Find non-EAX dword starting at ES:[(E)DI] |
| F2 A6 | REPNE CMPS m8,m8 | 5+9*N | Find matching bytes in ES:[(E)DI] and [(E)SI] |
| F2 A7 | REPNE CMPS m16,m16 | 5+9*N | Find matching words in ES:[(E)DI] and [(E)SI] |
| F2 A7 | REPNE CMPS m32,m32 | 5+9*N | Find matching dwords in ES:[(E)DI] and [(E)SI] |
| F2 AE | REPNE SCAS m8 | 5+8*N | Find AL, starting at ES:[(E)DI] |
| F2 AF | REPNE SCAS m16 | 5+8*N | Find AX, starting at ES:[(E)DI] |
| F2 AF | REPNE SCAS m32 | 5+8*N | Find EAX, starting at ES:[(E)DI] |

Operation

```
IF AddressSize = 16
THEN use CX for CountReg;
ELSE (* AddressSize = 32 *) use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
DO
   service pending interrupts (if any);
   perform primitive string instruction;
```

```
   CountReg ← CountReg - 1;
   IF primitive operation is CMPB, CMPW, SCAB, or SCAW
   THEN
      IF (instruction is REP/REPE/REPZ) AND (ZF=1)
      THEN exit WHILE loop
      ELSE
         IF (instruction is REPNZ or REPNE) AND (ZF=0)
         THEN exit WHILE loop;
         FI;
      FI;
   FI;
OD;
```

Description

REP, REPE (repeat while equal), and REPNE (repeat while not equal) are prefix that are applied to string operation. Each prefix cause the string instruction that follows to be repeated the number of times indicated in the count register or (for REPE and REPNE) until the indicated condition in the zero flag is no longer met.

Synonymous forms of REPE and REPNE are REPZ and REPNZ, respectively.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

The precise action for each iteration is as follows:

1. If the address-size attribute is 16 bits, use CX for the count register; if the address-size attribute is 32 bits, use ECX for the count register.

2. Check CX. If it is zero, exit the iteration, and move to the next instruction.

3. Acknowledge any pending interrupts.

4. Perform the string operation once.

5. Decrement CX or ECX by one; no flags are modified.

6. Check the zero flag if the string operation is SCAS or CMPS. If the repeat condition does not hold, exit the iteration and move to the next instruction. Exit the iteration if the prefix is REPE and ZF is 0 (the last comparison was not equal), or if the prefix is REPNE and ZF is one (the last comparison was equal).

7. Return to step 1 for the next iteration.

Repeated CMPS and SCAS instructions can be exited if the count is exhausted or if the zero flag fails the repeat condition. These two cases can be distinguished by using either the JCXZ instruction, or by using the conditional jumps that test the zero flag (JZ, JNZ, and JNE).

Flags Affected

ZF by REP CMPS and REP SCAS as described above

Protected Mode Exceptions

#UD if a repeat prefix is used before an instruction that is not in the list above; further exceptions can be generated when the string operation is executed; refer to the descriptions of the string instructions themselves

Real Address Mode Exceptions

Interrupt 6 if a repeat prefix is used before an instruction that is not in the list above; further exceptions can be generated when the string operation is executed; refer to the descriptions of the string instructions themselves

Virtual 8086 Mode Exceptions

#UD if a repeat prefix is used before an instruction that is not in the list above; further exceptions can be generated when the string operation is executed; refer to the descriptions of the string instructions themselves

Notes

Not all input/output ports can handle the rate at which the REP INS and REP OUTS instructions execute.

**RET — Return from Procedure**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| C3 | RET | 10+m | Return (near) to caller |
| CB | RET | 18+m,pm=32+m | Return (far) to caller, same privilege |
| CB | RET | pm=68 | Return (far), lesser privilege, switch stacks |
| C2 iw | RET imm16 | 10+m | Return (near), pop imm16 bytes of parameters |
| CA iw | RET imm16 | 18+m,pm=32+m | Return (far), same privilege, pop imm16 bytes |
| CA iw | RET imm16 | pm=68 | Return (far), lesser privilege, pop imm16 bytes |

Operation

```
IF instruction = near RET
THEN;
   IF OperandSize = 16
   THEN
      IP ← Pop();
      EIP ← EIP AND 0000FFFFH;
   ELSE (* OperandSize = 32 *)
      EIP ← Pop();
   FI;
   IF instruction has immediate operand THEN eSP ← eSP + imm16; FI;
FI;

IF (PE = 0 OR (PE = 1 AND VM = 1))
   (* real mode or virtual 8086 mode *)
   AND instruction = far RET
THEN;
   IF OperandSize = 16
   THEN
      IP ← Pop();
      EIP ← EIP AND 0000FFFFH;
      CS ← Pop(); (* 16-bit pop *)
   ELSE (* OperandSize = 32 *)
      EIP ← Pop();
      CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
   FI;
   IF instruction has immediate operand THEN eSP ← eSP + imm16; FI;
FI;

IF (PE = 1 AND VM = 0) (* Protected mode, not V86 mode *)
   AND instruction = far RET
THEN
   IF OperandSize=32
   THEN Third word on stack must be within stack limits else #SS(0);
   ELSE Second word on stack must be within stack limits else #SS(0);
   FI;
   Return selector RPL must be ≥ CPL ELSE #GP(return selector)
   IF return selector RPL = CPL
   THEN GOTO SAME-LEVEL;
   ELSE GOTO OUTER-PRIVILEGE-LEVEL;
   FI;
FI;
```

```
SAME-LEVEL:
   Return selector must be non-null ELSE #GP(0)
   Selector index must be within its descriptor table limits ELSE
      #GP(selector)
   Descriptor AR byte must indicate code segment ELSE #GP(selector)
   IF non-conforming
   THEN code segment DPL must equal CPL;
   ELSE #GP(selector);
   FI;
   IF conforming
   THEN code segment DPL must be ≤ CPL;
   ELSE #GP(selector);
   FI;
   Code segment must be present ELSE #NP(selector);
   Top word on stack must be within stack limits ELSE #SS(0);
   IP must be in code segment limit ELSE #GP(0);
   IF OperandSize=32
   THEN
      Load CS:EIP from stack
      Load CS register with descriptor
      Increment eSP by 8 plus the immediate offset if it exists
   ELSE (* OperandSize=16 *)
      Load CS:IP from stack
      Load CS register with descriptor
      Increment eSP by 4 plus the immediate offset if it exists
   FI;

OUTER-PRIVILEGE-LEVEL:
   IF OperandSize=32
   THEN Top (16+immediate) bytes on stack must be within stack limits
      ELSE #SS(0);
   ELSE Top (8+immediate) bytes on stack must be within stack limits ELSE
      #SS(0);
   FI;
   Examine return CS selector and associated descriptor:
      Selector must be non-null ELSE #GP(0);
      Selector index must be within its descriptor table limits ELSE
         #GP(selector)
      Descriptor AR byte must indicate code segment ELSE #GP(selector);
      IF non-conforming
      THEN code segment DPL must equal return selector RPL
      ELSE #GP(selector);
      FI;
      IF conforming
      THEN code segment DPL must be ≤ return selector RPL;
      ELSE #GP(selector);
      FI;
      Segment must be present ELSE #NP(selector)
   Examine return SS selector and associated descriptor:
      Selector must be non-null ELSE #GP(0);
      Selector index must be within its descriptor table limits
         ELSE #GP(selector);
      Selector RPL must equal the RPL of the return CS selector ELSE
         #GP(selector);
      Descriptor AR byte must indicate a writable data segment ELSE
         #GP(selector);
      Descriptor DPL must equal the RPL of the return CS selector ELSE
```

```
      #GP(selector);
   Segment must be present ELSE #NP(selector);
IP must be in code segment limit ELSE #GP(0);
Set CPL to the RPL of the return CS selector;
IF OperandMode=32
THEN
   Load CS:EIP from stack;
   Set CS RPL to CPL;
   Increment eSP by 8 plus the immediate offset if it exists;
   Load SS:eSP from stack;
ELSE (* OperandMode=16 *)
   Load CS:IP from stack;
   Set CS RPL to CPL;
   Increment eSP by 4 plus the immediate offset if it exists;
   Load SS:eSP from stack;
FI;
Load the CS register with the return CS descriptor;
Load the SS register with the return SS descriptor;
For each of ES, FS, GS, and DS
DO
   IF the current register setting is not valid for the outer level,
      set the register to null (selector ← AR ← 0);
   To be valid, the register setting must satisfy the following
      properties:
      Selector index must be within descriptor table limits;
      Descriptor AR byte must indicate data or readable code segment;
      IF segment is data or non-conforming code, THEN
         DPL must be ≥ CPL, or DPL must be ≥ RPL;
   FI;
OD;
```

Description

RET transfers control to a return address located on the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL.

The optional numeric parameter to RET gives the number of stack bytes (OperandMode=16) or words (OperandMode=32) to be released after the return address is popped. These items are typically used as input parameters to the procedure called.

For the intrasegment (near) return, the address on the stack is a segment offset, which is popped into the instruction pointer. The CS register is unchanged. For the intersegment (far) return, the address on the stack is a long pointer. The offset is popped first, followed by the selector.

In real mode, CS and IP are loaded directly. In Protected Mode, an intersegment return causes the processor to check the descriptor addressed by the return selector. The AR byte of the descriptor must indicate a code segment of equal or lesser privilege (or greater or equal numeric value) than the current privilege level. Returns to a lesser privilege level cause the stack to be reloaded from the value saved beyond the parameter block.

The DS, ES, FS, and GS segment registers can be set to 0 by the RET instruction during an interlevel transfer. If these registers refer to segments that cannot be used by the new privilege level, they are set to 0 to prevent unauthorized access from the new privilege level.

Flags Affected

None

Protected Mode Exceptions

#GP, #NP, or #SS, as described under "Operation" above; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would be outside the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## SAHF —— Store AH into Flags

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 9E | SAHF | 3 | Store AH into flags SF ZF xx AF xx PF xx CF |

### Operation

SF:ZF:xx:AF:xx:PF:xx:CF ← AH;

### Description

SAHF loads the flags listed above with values from the AH register, from bits 7, 6, 4, 2, and 0, respectively.

### Flags Affected

SF, ZF, AF, PF, and CF as described above

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

**SAL/SAR/SHL/SHR —— Shift Instructions**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| D0 /4 | SAL r/m8,1 | 3/7 | Multiply r/m byte by 2, once |
| D2 /4 | SAL r/m8,CL | 3/7 | Multiply r/m byte by 2, CL times |
| C0 /4 ib | SAL r/m8,imm8 | 3/7 | Multiply r/m byte by 2, imm8 times |
| D1 /4 | SAL r/m16,1 | 3/7 | Multiply r/m word by 2, once |
| D3 /4 | SAL r/m16,CL | 3/7 | Multiply r/m word by 2, CL times |
| C1 /4 ib | SAL r/m16,imm8 | 3/7 | Multiply r/m word by 2, imm8 times |
| D1 /4 | SAL r/m32,1 | 3/7 | Multiply r/m dword by 2, once |
| D3 /4 | SAL r/m32,CL | 3/7 | Multiply r/m dword by 2, CL times |
| C1 /4 ib | SAL r/m32,imm8 | 3/7 | Multiply r/m dword by 2, imm8 times |
| D0 /7 | SAR r/m8,1 | 3/7 | Signed divide^(1) r/m byte by 2, once |
| D2 /7 | SAR r/m8,CL | 3/7 | Signed divide^(1) r/m byte by 2, CL times |
| C0 /7 ib | SAR r/m8,imm8 | 3/7 | Signed divide^(1) r/m byte by 2, imm8 times |
| D1 /7 | SAR r/m16,1 | 3/7 | Signed divide^(1) r/m word by 2, once |
| D3 /7 | SAR r/m16,CL | 3/7 | Signed divide^(1) r/m word by 2, CL times |
| C1 /7 ib | SAR r/m16,imm8 | 3/7 | Signed divide^(1) r/m word by 2, imm8 times |
| D1 /7 | SAR r/m32,1 | 3/7 | Signed divide^(1) r/m dword by 2, once |
| D3 /7 | SAR r/m32,CL | 3/7 | Signed divide^(1) r/m dword by 2, CL times |
| C1 /7 ib | SAR r/m32,imm8 | 3/7 | Signed divide^(1) r/m dword by 2, imm8 times |
| D0 /4 | SHL r/m8,1 | 3/7 | Multiply r/m byte by 2, once |
| D2 /4 | SHL r/m8,CL | 3/7 | Multiply r/m byte by 2, CL times |
| C0 /4 ib | SHL r/m8,imm8 | 3/7 | Multiply r/m byte by 2, imm8 times |
| D1 /4 | SHL r/m16,1 | 3/7 | Multiply r/m word by 2, once |
| D3 /4 | SHL r/m16,CL | 3/7 | Multiply r/m word by 2, CL times |
| C1 /4 ib | SHL r/m16,imm8 | 3/7 | Multiply r/m word by 2, imm8 times |
| D1 /4 | SHL r/m32,1 | 3/7 | Multiply r/m dword by 2, once |
| D3 /4 | SHL r/m32,CL | 3/7 | Multiply r/m dword by 2, CL times |
| C1 /4 ib | SHL r/m32,imm8 | 3/7 | Multiply r/m dword by 2, imm8 times |
| D0 /5 | SHR r/m8,1 | 3/7 | Unsigned divide r/m byte by 2, once |
| D2 /5 | SHR r/m8,CL | 3/7 | Unsigned divide r/m byte by 2, CL times |
| C0 /5 ib | SHR r/m8,imm8 | 3/7 | Unsigned divide r/m byte by 2, imm8 times |
| D1 /5 | SHR r/m16,1 | 3/7 | Unsigned divide r/m word by 2, once |
| D3 /5 | SHR r/m16,CL | 3/7 | Unsigned divide r/m word by 2, CL times |
| C1 /5 ib | SHR r/m16,imm8 | 3/7 | Unsigned divide r/m word by 2, imm8 times |
| D1 /5 | SHR r/m32,1 | 3/7 | Unsigned divide r/m dword by 2, once |
| D3 /5 | SHR r/m32,CL | 3/7 | Unsigned divide r/m dword by 2, CL times |
| C1 /5 ib | SHR r/m32,imm8 | 3/7 | Unsigned divide r/m dword by 2, imm8 times |

Not the same division as IDIV; rounding is toward negative infinity.

Operation

```
(* COUNT is the second parameter *)
(temp) ← COUNT;
WHILE (temp ≠ 0)
DO
   IF instruction is SAL or SHL
   THEN CF ← high-order bit of r/m;
   FI;
   IF instruction is SAR or SHR
   THEN CF ← low-order bit of r/m;
   FI;
   IF instruction = SAL or SHL
   THEN r/m ← r/m * 2;
   FI;
   IF instruction = SAR
   THEN r/m ← r/m /2 (*Signed divide, rounding toward negative infinity*);
   FI;
   IF instruction = SHR
```

```
   THEN r/m ← r/m / 2; (* Unsigned divide *);
   FI;
   temp ← temp - 1;
OD;
(* Determine overflow for the various instructions *)
IF COUNT = 1
THEN
   IF instruction is SAL or SHL
   THEN OF ← high-order bit of r/m ≠ (CF);
   FI;
   IF instruction is SAR
   THEN OF ← 0;
   FI;
   IF instruction is SHR
   THEN OF ← high-order bit of operand;
   FI;
ELSE OF ← undefined;
FI;
```

Description

SAL (or its synonym, SHL) shifts the bits of the operand upward. The high-order bit is shifted into the carry flag, and the low-order bit is set to 0.

SAR and SHR shift the bits of the operand downward. The low-order bit is shifted into the carry flag. The effect is to divide the operand by 2. SAR performs a signed divide with rounding toward negative infinity (not the same as IDIV); the high-order bit remains the same. SHR performs an unsigned divide; the high-order bit is set to 0.

The shift is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum execution time, the 80386 does not allow shift counts greater than 31. If a shift count greater than 31 is attempted, only the bottom five bits of the shift count are used. (The 8086 uses all eight bits of the shift count.)

The overflow flag is set only if the single-shift forms of the instructions are used. For left shifts, OF is set to 0 if the high bit of the answer is the same as the result of the carry flag (i.e., the top two bits of the original operand were the same); OF is set to 1 if they are different. For SAR, OF is set to 0 for all single shifts. For SHR, OF is set to the high-order bit of the original operand.

Flags Affected

OF for single shifts; OF is undefined for multiple shifts; CF, ZF, PF, and SF as described in Appendix C

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**SBB —— Integer Subtraction with Borrow**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 1C ib | SBB AL,imm8 | 2 | Subtract with borrow immediate byte from AL |
| 1D iw | SBB AX,imm16 | 2 | Subtract with borrow immediate word from AX |
| 1D id | SBB EAX,imm32 | 2 | Subtract with borrow immediate dword from EAX |
| 80 /3 ib | SBB r/m8,imm8 | 2/7 | Subtract with borrow immediate byte from r/m byte |
| 81 /3 iw | SBB r/m16,imm16 | 2/7 | Subtract with borrow immediate from r/m word |
| 81 /3 id | SBB r/m32,imm32 | 2/7 | Subtract with borrow immediate dword from r/m dword |
| 83 /3 ib | SBB r/m16,imm8 | 2/7 | Subtract with borrow sign-extended immediate byte from r/m word |
| 83 /3 ib | SBB r/m32,imm8 | 2/7 | Subtract with borrow sign-extended immediate byte from r/m dword |
| 18 /r | SBB r/m8,r8 | 2/6 | Subtract with borrow byte register from r/m byte |
| 19 /r | SBB r/m16,r16 | 2/6 | Subtract with borrow word register from r/m word |
| 19 /r | SBB r/m32,r32 | 2/6 | Subtract with borrow dword from r/m dword |
| 1A /r | SBB r8,r/m8 | 2/7 | Subtract with borrow byte register from r/m byte |
| 1B /r | SBB r16,r/m16 | 2/7 | Subtract with borrow word register from r/m word |
| 1B /r | SBB r32,r/m32 | 2/7 | Subtract with borrow dword register from r/m dword |

Operation

```
IF SRC is a byte and DEST is a word or dword
THEN DEST = DEST - (SignExtend(SRC) + CF)
ELSE DEST ← DEST - (SRC + CF);
```

Description

SBB adds the second operand (DEST) to the carry flag (CF) and subtracts the result from the first operand (SRC). The result of the subtraction is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended.

Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix C

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## SCAS/SCASB/SCASW/SCASD —— Compare String Data

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| AE | SCAS m8 | 7 | Compare bytes AL-ES:[DI], update (E)DI |
| AF | SCAS m16 | 7 | Compare words AX-ES:[DI], update (E)DI |
| AF | SCAS m32 | 7 | Compare dwords EAX-ES:[DI], update (E)DI |
| AE | SCASB | 7 | Compare bytes AL-ES:[DI], update (E)DI |
| AF | SCASW | 7 | Compare words AX-ES:[DI], update (E)DI |
| AF | SCASD | 7 | Compare dwords EAX-ES:[DI], update (E)DI |

### Operation

```
IF AddressSize = 16
THEN use DI for dest-index;
ELSE (* AddressSize = 32 *) use EDI for dest-index;
FI;
IF byte type of instruction
THEN
   AL - [dest-index]; (* Compare byte in AL and dest *)
   IF DF = 0 THEN IndDec ← 1 ELSE IncDec ← -1; FI;
ELSE
   IF OperandSize = 16
   THEN
      AX - [dest-index]; (* compare word in AL and dest *)
      IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← -2; FI;
   ELSE (* OperandSize = 32 *)
      EAX - [dest-index];(* compare dword in EAX & dest *)
      IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← -4; FI;
   FI;
FI;
dest-index = dest-index + IncDec
```

### Description

SCAS subtracts the memory byte or word at the destination register from the AL, AX or EAX register. The result is discarded; only the flags are set. The operand must be addressable from the ES segment; no segment override is possible.

If the address-size attribute for this instruction is 16 bits, DI is used as the destination register; otherwise, the address-size attribute is 32 bits and EDI is used.

The address of the memory data being compared is determined solely by the contents of the destination register, not by the operand to SCAS. The operand validates ES segment addressability and determines the data type. Load the correct index value into DI or EDI before executing SCAS.

After the comparison is made, the destination register is automatically updated. If the direction flag is 0 (CLD was executed), the destination register is incremented; if the direction flag is 1 (STD was executed), it is decremented. The increments or decrements are by 1 if bytes are compared, by 2 if words are compared, or by 4 if doublewords are compared.

SCASB, SCASW, and SCASD are synonyms for the byte, word and doubleword SCAS instructions that don't require operands. They are simpler to code, but provide no type or segment checking.

SCAS can be preceded by the REPE or REPNE prefix for a block search of CX or ECX bytes or words. Refer to the REP instruction for further details.

Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix C

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## SETcc —— Byte Set on Condition

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 97 | SETA r/m8 | 4/5 | Set byte if above (CF=0 and ZF=0) |
| 0F 93 | SETAE r/m8 | 4/5 | Set byte if above or equal (CF=0) |
| 0F 92 | SETB r/m8 | 4/5 | Set byte if below (CF=1) |
| 0F 96 | SETBE r/m8 | 4/5 | Set byte if below or equal (CF=1 or (ZF=1) |
| 0F 92 | SETC r/m8 | 4/5 | Set if carry (CF=1) |
| 0F 94 | SETE r/m8 | 4/5 | Set byte if equal (ZF=1) |
| 0F 9F | SETG r/m8 | 4/5 | Set byte if greater (ZF=0 or SF=OF) |
| 0F 9D | SETGE r/m8 | 4/5 | Set byte if greater or equal (SF=OF) |
| 0F 9C | SETL r/m8 | 4/5 | Set byte if less (SF≠OF) |
| 0F 9E | SETLE r/m8 | 4/5 | Set byte if less or equal (ZF=1 and SF≠OF) |
| 0F 96 | SETNA r/m8 | 4/5 | Set byte if not above (CF=1) |
| 0F 92 | SETNAE r/m8 | 4/5 | Set byte if not above or equal (CF=1) |
| 0F 93 | SETNB r/m8 | 4/5 | Set byte if not below (CF=0) |
| 0F 97 | SETNBE r/m8 | 4/5 | Set byte if not below or equal (CF=0 and ZF=0) |
| 0F 93 | SETNC r/m8 | 4/5 | Set byte if not carry (CF=0) |
| 0F 95 | SETNE r/m8 | 4/5 | Set byte if not equal (ZF=0) |
| 0F 9E | SETNG r/m8 | 4/5 | Set byte if not greater (ZF=1 or SF≠OF) |
| 0F 9C | SETNGE r/m8 | 4/5 | Set if not greater or equal (SF≠OF) |
| 0F 9D | SETNL r/m8 | 4/5 | Set byte if not less (SF=OF) |
| 0F 9F | SETNLE r/m8 | 4/5 | Set byte if not less or equal (ZF=1 and SF≠OF) |
| 0F 91 | SETNO r/m8 | 4/5 | Set byte if not overflow (OF=0) |
| 0F 9B | SETNP r/m8 | 4/5 | Set byte if not parity (PF=0) |
| 0F 99 | SETNS r/m8 | 4/5 | Set byte if not sign (SF=0) |
| 0F 95 | SETNZ r/m8 | 4/5 | Set byte if not zero (ZF=0) |
| 0F 90 | SETO r/m8 | 4/5 | Set byte if overflow (OF=1) |
| 0F 9A | SETP r/m8 | 4/5 | Set byte if parity (PF=1) |
| 0F 9A | SETPE r/m8 | 4/5 | Set byte if parity even (PF=1) |
| 0F 9B | SETPO r/m8 | 4/5 | Set byte if parity odd (PF=0) |
| 0F 98 | SETS r/m8 | 4/5 | Set byte if sign (SF=1) |
| 0F 94 | SETZ r/m8 | 4/5 | Set byte if zero (ZF=1) |

### Operation

```
IF condition THEN r/m8 ← 1 ELSE r/m8 ← 0; FI;
```

### Description

SETcc stores a byte at the destination specified by the effective address or register if the condition is met, or a 0 byte if the condition is not met.

### Flags Affected

None

### Protected Mode Exceptions

#GP(0) if the result is in a non-writable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## SGDT/SIDT —— Store Global/Interrupt Descriptor Table Register

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 01 /0 | SGDT m | 9 | Store GDTR to m |
| 0F 01 /1 | SIDT m | 9 | Store IDTR to m |

Operation

DEST ← 48-bit BASE/LIMIT register contents;

Description

SGDT/SIDT copies the contents of the descriptor table register the six bytes of memory indicated by the operand. The LIMIT field of the register is assigned to the first word at the effective address. If the operand-size attribute is 32 bits, the next three bytes are assigned the BASE field of the register, and the fourth byte is written with zero. The last byte is undefined. Otherwise, if the operand-size attribute is 16 bits, the next 4 bytes are assigned the 32-bit BASE field of the register.

SGDT and SIDT are used only in operating system software; they are not used in application programs.

Flags Affected

None

Protected Mode Exceptions

Interrupt 6 if the destination operand is a register; #GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 6 if the destination operand is a register; Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

Compatability Note

The 16-bit forms of the SGDT/SIDT instructions are compatible with the 80286, if the value in the upper eight bits is not referenced. The 80286 stores 1's in these upper bits, whereas the 80386 stores 0's if the operand-size attribute is 16 bits. These bits were specified as undefined by the SGDT/SIDT instructions in the iAPX 286 Programmer's Reference Manual.

**SHLD —— Double Precision Shift Left**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F A4 | SHLD r/m16,r16,imm8 | 3/7 | r/m16 gets SHL of r/m16 concatenated with r16 |
| 0F A4 | SHLD r/m32,r32,imm8 | 3/7 | r/m32 gets SHL of r/m32 concatenated with r32 |
| 0F A5 | SHLD r/m16,r16,CL | 3/7 | r/m16 gets SHL of r/m16 concatenated with r16 |
| 0F A5 | SHLD r/m32,r32,CL | 3/7 | r/m32 gets SHL of r/m32 concatenated with r32 |

Operation

```
(* count is an unsigned integer corresponding to the last operand of the
instruction, either an immediate byte or the byte in register CL *)
ShiftAmt ← count MOD 32;
inBits ← register; (* Allow overlapped operands *)
IF ShiftAmt = 0
THEN no operation
ELSE
   IF ShiftAmt ≥ OperandSize
   THEN (* Bad parameters *)
      r/m ← UNDEFINED;
      CF, OF, SF, ZF, AF, PF ← UNDEFINED;
   ELSE (* Perform the shift *)
      CF ← BIT[Base, OperandSize - ShiftAmt];
         (* Last bit shifted out on exit *)
   FOR i ← OperandSize - 1 DOWNTO ShiftAmt
   DO
      BIT[Base, i] ← BIT[Base, i - ShiftAmt];
   OF;
   FOR i ← ShiftAmt - 1 DOWNTO 0
   DO
      BIT[Base, i] ← BIT[inBits, i - ShiftAmt + OperandSize];
   OD;
   Set SF, ZF, PF (r/m);
      (* SF, ZF, PF are set according to the value of the result *)
   AF ← UNDEFINED;
   FI;
FI;
```

Description

SHLD shifts the first operand provided by the r/m field to the left as many bits as specified by the count operand. The second operand (r16 or r32) provides the bits to shift in from the right (starting with bit 0). The result is stored back into the r/m operand. The register remains unaltered.

The count operand is provided by either an immediate byte or the contents of the CL register. These operands are taken MODULO 32 to provide a number between 0 and 31 by which to shift. Because the bits to shift are provided by the specified registers, the operation is useful for multiprecision shifts (64 bits or more). The SF, ZF and PF flags are set according to the value of the result. CS is set to the value of the last bit shifted out. OF and AF are left undefined.

Flags Affected

OF, SF, ZF, PF, and CF as described above; AF and OF are undefined

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**SHRD —— Double Precision Shift Right**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F AC | SHRD r/m16,r16,imm8 | 3/7 | r/m16 gets SHR of r/m16 concatenated with r16 |
| 0F AC | SHRD r/m32,r32,imm8 | 3/7 | r/m32 gets SHR of r/m32 with r32 |
| 0F AD | SHRD r/m16,r16,CL | 3/7 | r/m16 gets SHR of r/m16 concatenated with r16 |
| 0F AD | SHRD r/m32,r32,CL | 3/7 | r/m32 gets SHR of r/m32 concatenated with r32 |

Operation

```
(* count is an unsigned integer corresponding to the last operand of the
instruction, either an immediate byte or the byte in register CL *)
ShiftAmt ← count MOD 32;
inBits ← register; (* Allow overlapped operands *)
IF ShiftAmt = 0
THEN no operation
ELSE
   IF ShiftAmt ≥ OperandSize
   THEN (* Bad parameters *)
      r/m ← UNDEFINED;
      CF, OF, SF, ZF, AF, PF ← UNDEFINED;
   ELSE (* Perform the shift *)
      CF ← BIT[r/m, ShiftAmt - 1]; (* last bit shifted out on exit *)
      FOR i ← 0 TO OperandSize - 1 - ShiftAmt
      DO
         BIT[r/m, i] ← BIT[r/m, i - ShiftAmt];
      OD;
      FOR i ← OperandSize - ShiftAmt TO OperandSize - 1
      DO
         BIT[r/m,i] ← BIT[inBits,i+ShiftAmt - OperandSize];
      OD;
      Set SF, ZF, PF (r/m);
         (* SF, ZF, PF are set according to the value of the result *)
      Set SF, ZF, PF (r/m);
      AF ← UNDEFINED;
   FI;
FI;
```

Description

SHRD shifts the first operand provided by the r/m field to the right as many bits as specified by the count operand. The second operand (r16 or r32) provides the bits to shift in from the left (starting with bit 31). The result is stored back into the r/m operand. The register remains unaltered.

The count operand is provided by either an immediate byte or the contents of the CL register. These operands are taken MODULO 32 to provide a number between 0 and 31 by which to shift. Because the bits to shift are provided by the specified register, the operation is useful for multi-precision shifts (64 bits or more). The SF, ZF and PF flags are set according to the value of the result. CS is set to the value of the last bit shifted out. OF and AF are left undefined.

Flags Affected

OF, SF, ZF, PF, and CF as described above; AF and OF are undefined

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## SLDT — Store Local Descriptor Table Register

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 00 /0 | SLDT r/m16 | pm=2/2 | Store LDTR to EA word |

### Operation

r/m16 ← LDTR;

### Description

SLDT stores the Local Descriptor Table Register (LDTR) in the two-byte register or memory location indicated by the effective address operand. This register is a selector that points into the Global Descriptor Table.

SLDT is used only in operating system software. It is not used in application programs.

### Flags Affected

None

### Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

### Real Address Mode Exceptions

Interrupt 6; SLDT is not recognized in Real Address Mode

### Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

### Notes

The operand-size attribute has no effect on the operation of the instruction.

## SMSW — Store Machine Status Word

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 01 /4 | SMSW r/m16 | 2/3,pm=2/2 | Store machine status word to EA word |

### Operation

r/m16 ← MSW;

### Description

SMSW stores the machine status word (part of CR0) in the two-byte register or memory location indicated by the effective address operand.

### Flags Affected

None

### Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

### Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

### Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

### Notes

This instruction is provided for compatibility with the 80286; 80386 programs should use MOV ..., CR0.

## STC — Set Carry Flag

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F9 | STC | 2 | Set carry flag |

### Operation

```
CF ← 1;
```

### Description

STC sets the carry flag to 1.

### Flags Affected

CF = 1

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

## STD — Set Direction Flag

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| FD | STD | 2 | Set direction flag so (E)SI and/or (E)DI decrement |

### Operation

```
DF ← 1;
```

### Description

STD sets the direction flag to 1, causing all subsequent string operations to decrement the index registers, (E)SI and/or (E)DI, on which they operate.

### Flags Affected

DF = 1

### Protected Mode Exceptions

None

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

## STI —— Set Interrupt Flag

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| F13 | STI | 3 | Set interrupt flag; interrupts enabled at the end of the next instruction |

### Operation

IF ← 1

### Description

STI sets the interrupt flag to 1. The 80386 then responds to external interrupts after executing the next instruction if the next instruction allows the interrupt flag to remain enabled. If external interrupts are disabled and you code STI, RET (such as at the end of a subroutine), the RET is allowed to execute before external interrupts are recognized. Also, if external interrupts are disabled and you code STI, CLI, then external interrupts are not recognized because the CLI instruction clears the interrupt flag during its execution.

### Flags Affected

IF = 1

### Protected Mode Exceptions

#GP(0) if the current privilege level is greater (has less privilege) than the I/O privilege level

### Real Address Mode Exceptions

None

### Virtual 8086 Mode Exceptions

None

## STOS/STOSB/STOSW/STOSD —— Store String Data

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| AA | STOS m8 | 4 | Store AL in byte ES:[(E)DI], update (E)DI |
| AB | STOS m16 | 4 | Store AX in word ES:[(E)DI], update (E)DI |
| AB | STOS m32 | 4 | Store EAX in dword ES:[(E)DI], update (E)DI |
| AA | STOSB | 4 | Store AL in byte ES:[(E)DI], update (E)DI |
| AB | STOSW | 4 | Store AX in word ES:[(E)DI], update (E)DI |
| AB | STOSD | 4 | Store EAX in dword ES:[(E)DI], update (E)DI |

Operation

```
IF AddressSize = 16
THEN use ES:DI for DestReg
ELSE (* AddressSize = 32 *) use ES:EDI for DestReg;
FI;
IF byte type of instruction
THEN
   (ES:DestReg) ← AL;
   IF DF = 0
   THEN DestReg ← DestReg + 1;
   ELSE DestReg ← DestReg - 1;
   FI;
ELSE IF OperandSize = 16
   THEN
      (ES:DestReg) ← AX;
      IF DF = 0
      THEN DestReg ← DestReg + 2;
      ELSE DestReg ← DestReg - 2;
      FI;
   ELSE (* OperandSize = 32 *)
      (ES:DestReg) ← EAX;
      IF DF = 0
      THEN DestReg ← DestReg + 4;
      ELSE DestReg ← DestReg - 4;
      FI;
   FI;
FI;
```

Description

STOS transfers the contents of all AL, AX, or EAX register to the memory byte or word given by the destination register relative to the ES segment. The destination register is DI for an address-size attribute of 16 bits or EDI for an address-size attribute of 32 bits.

The destination operand must be addressable from the ES register. A segment override is not possible.

The address of the destination is determined by the contents of the destination register, not by the explicit operand of STOS. This operand is used only to validate ES segment addressability and to determine the data type. Load the correct index value into the destination register before executing STOS.

After the transfer is made, DI is automatically updated. If the direction flag is 0 (CLD was executed), DI is incremented; if the direction flag is 1 (STD was executed), DI is decremented. DI is incremented or decremented by 1 if a byte is stored, by 2 if a word is stored, or by 4 if a doubleword is stored.

STOSB, STOSW, and STOSD are synonyms for the byte, word, and doubleword STOS instructions, that do not require an operand. They are simpler to use, but provide no type or segment checking.

STOS can be preceded by the REP prefix for a block fill of CX or ECX bytes, words, or doublewords. Refer to the REP instruction for further details.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## STR —— Store Task Register

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 00 /1 | STR r/m16 | pm=23/27 | Load EA word into task register |

### Operation

r/m ← task register;

### Description

The contents of the task register are copied to the two-byte register or memory location indicated by the effective address operand.

STR is used only in operating system software. It is not used in application programs.

### Flags Affected

None

### Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

### Real Address Mode Exceptions

Interrupt 6; STR is not recognized in Real Address Mode

### Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode

### Notes

The operand-size attribute has no effect on this instruction.

**SUB —— Integer Subtraction**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 2C ib | SUB AL,imm8 | 2 | Subtract immediate byte from AL |
| 2D iw | SUB AX,imm16 | 2 | Subtract immediate word from AX |
| 2D id | SUB EAX,imm32 | 2 | Subtract immediate dword from EAX |
| 80 /5 ib | SUB r/m8,imm8 | 2/7 | Subtract immediate byte from r/m byte |
| 81 /5 iw | SUB r/m16,imm16 | 2/7 | Subtract immediate word from r/m word |
| 81 /5 id | SUB r/m32,imm32 | 2/7 | Subtract immediate dword from r/m dword |
| 83 /5 ib | SUB r/m16,imm8 | 2/7 | Subtract sign-extended immediate byte from r/m word |
| 83 /5 ib | SUB r/m32,imm8 | 2/7 | Subtract sign-extended immediate byte from r/m dword |
| 28 /r | SUB r/m8,r8 | 2/6 | Subtract byte register from r/m byte |
| 29 /r | SUB r/m16,r16 | 2/6 | Subtract word register from r/m word |
| 29 /r | SUB r/m32,r32 | 2/6 | Subtract dword register from r/m dword |
| 2A /r | SUB r8,r/m8 | 2/7 | Subtract byte register from r/m byte |
| 2B /r | SUB r16,r/m16 | 2/7 | Subtract word register from r/m word |
| 2B /r | SUB r32,r/m32 | 2/7 | Subtract dword register from r/m dword |

Operation

```
IF SRC is a byte and DEST is a word or dword
THEN DEST = DEST - SignExtend(SRC);
ELSE DEST ← DEST - SRC;
FI;
```

Description

SUB subtracts the second operand (SRC) from the first operand (DEST). The first operand is assigned the result of the subtraction, and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended to the size of the destination operand.

Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix C

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**TEST —— Logical Compare**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| A8 ib | TEST AL,imm8 | 2 | AND immediate byte with AL |
| A9 iw | TEST AX,imm16 | 2 | AND immediate word with AX |
| A9 id | TEST EAX,imm32 | 2 | AND immediate dword with EAX |
| F6 /0 ib | TEST r/m8,imm8 | 2/5 | AND immediate byte with r/m byte |
| F7 /0 iw | TEST r/m16,imm16 | 2/5 | AND immediate word with r/m word |
| F7 /0 id | TEST r/m32,imm32 | 2/5 | AND immediate dword with r/m dword |
| 84 /r | TEST r/m8,r8 | 2/5 | AND byte register with r/m byte |
| 85 /r | TEST r/m16,r16 | 2/5 | AND word register with r/m word |
| 85 /r | TEST r/m32,r32 | 2/5 | AND dword register with r/m dword |

Operation

```
DEST : = LeftSRC AND RightSRC;
CF ← 0;
OF ← 0;
```

Description

TEST computes the bit-wise logical AND of its two operands. Each bit of the result is 1 if both of the corresponding bits of the operands are 1; otherwise, each bit is 0. The result of the operation is discarded and only the flags are modified.

Flags Affected

OF = 0, CF = 0; SF, ZF, and PF as described in Appendix C

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

## VERR, VERW —— Verify a Segment for Reading or Writing

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 0F 00 /4 | VERR r/m16 | pm=10/11 | Set ZF=1 if segment can be read, selector in r/m16 |
| 0F 00 /5 | VERW r/m16 | pm=15/16 | Set ZF=1 if segment can be written, selector in r/m16 |

Operation

```
IF segment with selector at (r/m) is accessible
   with current protection level
   AND ((segment is readable for VERR) OR
      (segment is writable for VERW))
THEN ZF ← 0;
ELSE ZF ← 1;
FI;
```

Description

The two-byte register or memory operand of VERR and VERW contains the value of a selector. VERR and VERW determine whether the segment denoted by the selector is reachable from the current privilege level and whether the segment is readable (VERR) or writable (VERW). If the segment is accessible, the zero flag is set to 1; if the segment is not accessible, the zero flag is set to 0. To set ZF, the following conditions must be met:

- The selector must denote a descriptor within the bounds of the table (GDT or LDT); the selector must be "defined."
- The selector must denote the descriptor of a code or data segment (not that of a task state segment, LDT, or a gate).
- For VERR, the segment must be readable. For VERW, the segment must be a writable data segment.
- If the code segment is readable and conforming, the descriptor privilege level (DPL) can be any value for VERR. Otherwise, the DPL must be greater than or equal to (have less or the same privilege as) both the current privilege level and the selector's RPL.

The validation performed is the same as if the segment were loaded into DS, ES, FS, or GS, and the indicated access (read or write) were performed. The zero flag receives the result of the validation. The selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

Flags Affected

ZF as described above

Protected Mode Exceptions

Faults generated by illegal addressing of the memory operand that contains the selector, the selector is not loaded into any segment register, and no faults attributable to the selector operand are generated

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 6; VERR and VERW are not recognized in Real Address Mode

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**WAIT —— Wait until BUSY# Pin is Inactive (HIGH)**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 9B | WAIT | 6 min. | Wait until BUSY pin is inactive (HIGH) |

Description

WAIT suspends execution of 80386 instructions until the BUSY# pin is inactive (high). The BUSY# pin is driven by the 80287 numeric processor extension.

Flags Affected

None

Protected Mode Exceptions

#NM if the task-switched flag in the machine status word (the lower 16 bits of register CR0) is set; #MF if the ERROR# input pin is asserted (i.e., the 80287 has detected an unmasked numeric error)

Real Address Mode Exceptions

Same exceptions as in Protected Mode

Virtual 8086 Mode Exceptions

Same exceptions as in Protected Mode

**XCHG —— Exchange Register/Memory with Register**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 90 + r | XCHG AX,r16 | 3 | Exchange word register with AX |
| 90 + r | XCHG r16,AX | 3 | Exchange word register with AX |
| 90 + r | XCHG EAX,r32 | 3 | Exchange dword register with EAX |
| 90 + r | XCHG r32,EAX | 3 | Exchange dword register with EAX |
| 86 /r | XCHG r/m8,r8 | 3 | Exchange byte register with EA byte |
| 86 /r | XCHG r8,r/m8 | 3/5 | Exchange byte register with EA byte |
| 87 /r | XCHG r/m16,r16 | 3 | Exchange word register with EA word |
| 87 /r | XCHG r16,r/m16 | 3/5 | Exchange word register with EA word |
| 87 /r | XCHG r/m32,r32 | 3 | Exchange dword register with EA dword |
| 87 /r | XCHG r32,r/m32 | 3/5 | Exchange dword register with EA dword |

Operation

temp ← DEST
DEST ← SRC
SRC ← temp

Description

XCHG exchanges two operands. The operands can be in either order. If a memory operand is involved, BUS LOCK is asserted for the duration of the exchange, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL.

Flags Affected

None

Protected Mode Exceptions

#GP(0) if either operand is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**XLAT/XLATB —— Table Look-up Translation**

```
D7    XLAT m8    5     Set AL to memory byte DS:[(E)BX + unsigned AL]
D7    XLATB      5     Set AL to memory byte DS:[(E)BX + unsigned AL]
```

Operation

```
IF AddressSize = 16
THEN
   AL ← (BX + ZeroExtend(AL))
ELSE (* AddressSize = 32 *)
   AL ← (EBX + ZeroExtend(AL));
FI;
```

Description

XLAT changes the AL register from the table index to the table entry. AL should be the unsigned index into a table addressed by DS:BX (for an address-size attribute of 16 bits) or DS:EBX (for an address-size attribute of 32 bits).

The operand to XLAT allows for the possibility of a segment override. XLAT uses the contents of BX even if they differ from the offset of the operand. The offset of the operand should have been moved intoBX/EBX with a previous instruction.

The no-operand form, XLATB, can be used if the BX/EBX table will always reside in the DS segment.

Flags Affected

None

Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

**XOR —— Logical Exclusive OR**

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 34 ib | XOR AL,imm8 | 2 | Exclusive-OR immediate byte to AL |
| 35 iw | XOR AX,imm16 | 2 | Exclusive-OR immediate word to AX |
| 35 id | XOR EAX,imm32 | 2 | Exclusive-OR immediate dword to EAX |
| 80 /6 ib | XOR r/m8,imm8 | 2/7 | Exclusive-OR immediate byte to r/m byte |
| 81 /6 iw | XOR r/m16,imm16 | 2/7 | Exclusive-OR immediate word to r/m word |
| 81 /6 id | XOR r/m32,imm32 | 2/7 | Exclusive-OR immediate dword to r/m dword |
| 83 /6 ib | XOR r/m16,imm8 | 2/7 | XOR sign-extended immediate byte with r/m word |
| 83 /6 ib | XOR r/m32,imm8 | 2/7 | XOR sign-extended immediate byte with r/m dword |
| 30 /r | XOR r/m8,r8 | 2/6 | Exclusive-OR byte register to r/m byte |
| 31 /r | XOR r/m16,r16 | 2/6 | Exclusive-OR word register to r/m word |
| 31 /r | XOR r/m32,r32 | 2/6 | Exclusive-OR dword register to r/m dword |
| 32 /r | XOR r8,r/m8 | 2/7 | Exclusive-OR byte register to r/m byte |
| 33 /r | XOR r16,r/m16 | 2/7 | Exclusive-OR word register to r/m word |
| 33 /r | XOR r32,r/m32 | 2/7 | Exclusive-OR dword register to r/m dword |

Operation

DEST ← LeftSRC XOR RightSRC
CF ← 0
OF ← 0

Description

XOR computes the exclusive OR of the two operands. Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same. The answer replaces the first operand.

Flags Affected

CF = 0, OF = 0; SF, ZF, and PF as described in Appendix C; AF is undefined

Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault

# Appendix A Opcode Map

---

The opcode tables that follow aid in interpreting 80386 object code. Use the high-order four bits of the opcode as an index to a row of the opcode table; use the low-order four bits as an index to a column of the table. If the opcode is 0FH, refer to the two-byte opcode table and use the second byte of the opcode to index the rows and columns of that table.

Key to Abbreviations

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

Codes for Addressing Method

A Direct address; the instruction has no modR/M byte; the address of the operand is encoded in the instruction; no base register, index register, or scaling factor can be applied; e.g., far JMP (EA).

C The reg field of the modR/M byte selects a control register; e.g., MOV (0F20, 0F22).

D The reg field of the modR/M byte selects a debug register; e.g., MOV (0F21,0F23).

E A modR/M byte follows the opcode and specifies the operand. The operand is either a general register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

F Flags Register.

G The reg field of the modR/M byte selects a general register; e.g., ADD (00).

I Immediate data. The value of the operand is encoded in subsequent bytes of the instruction.

J The instruction contains a relative offset to be added to the instruction pointer register; e.g., JMP short, LOOP.

M The modR/M byte may refer only to memory; e.g., BOUND, LES, LDS, LSS, LFS, LGS.

O The instruction has no modR/M byte; the offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied; e.g., MOV (A0-A3).

R The mod field of the modR/M byte may refer only to a general register; e.g., MOV (0F20-0F24, 0F26).

S The reg field of the modR/M byte selects a segment register; e.g., MOV (8C,8E).

T The reg field of the modR/M byte selects a test register; e.g., MOV (0F24,0F26).

X Memory addressed by DS:SI; e.g., MOVS, COMPS, OUTS, LODS, SCAS.

Y Memory addressed by ES:DI; e.g., MOVS, CMPS, INS, STOS.

Codes for Operant Type

a Two one-word operands in memory or two double-word operands in memory, depending on operand size attribute (used only by BOUND).

b Byte (regardless of operand size attribute)

c Byte or word, depending on operand size attribute.

d Double word (regardless of operand size attribute)

p 32-bit or 48-bit pointer, depending on operand size attribute.

s Six-byte pseudo-descriptor

v Word or double word, depending on operand size attribute.

w Word (regardless of operand size attribute)

Register Codes

When an operand is a specific register encoded in the opcode, the register is identified by its name; e.g., AX, CL, or ESI. The name of the register indicates whether the register is 32-, 16-, or 8-bits wide. A register identifier of the form eXX is used when the width of the register depends on the operand size attribute; for example, eAX indicates that the AX register is used when the operand size attribute is 16 and the EAX register is used when the operand size attribute is 32.

One-Byte Opcode Map

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | ADD | | | | | | PUSH | POP | OR | | | | | | PUSH | 2-byte |
| | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | AL,Ib | eAX,Iv | ES | ES | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | AL,Ib | eAX,Iv | CS | escape |
| 1 | ADC | | | | | | PUSH | POP | SBB | | | | | | PUSH | POP |
| | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | AL,Ib | eAX,Iv | SS | SS | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | AL,Ib | eAX,Iv | DS | DS |
| 2 | AND | | | | | | SEG | DAA | SUB | | | | | | SEG | DAS |
| | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | AL,Ib | eAX,Iv | =ES | | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | AL,Ib | eAX,Iv | =CS | |
| 3 | XOR | | | | | | SEG | AAA | CMP | | | | | | SEG | AAS |
| | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | AL,Ib | eAX,Iv | =SS | | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | AL,Ib | eAX,Iv | =CS | |
| 4 | INC general register | | | | | | | | DEC general register | | | | | | | |
| | eAX | eCX | eDX | eBX | eSP | eBP | eSI | eDI | eAX | eCX | eDX | eBX | eSP | eBP | eSI | eDI |
| 5 | PUSH general register | | | | | | | | POP into general register | | | | | | | |
| | eAX | eCX | eDX | eBX | eSP | eBP | eSI | eDI | eAX | eCX | eDX | eBX | eSP | eBP | eSI | eDI |
| 6 | PUSHA | POPA | BOUND Gv,Ma | ARPL Ew,Rw | SEG =FS | SEG =GS | Operand Size | Address Size | PUSH Ib | IMUL GvEvIv | PUSH Ib | IMUL GvEvIv | INSB Yb,DX | INSW/D Yb,DX | OUTSB Dx,Xb | OUTSW/D DX,Xv |
| 7 | Short displacement jump of condition (Jb) | | | | | | | | Short-displacement jump on condition(Jb) | | | | | | | |
| | JO | JNO | JB | JNB | JZ | JNZ | JBE | JNBE | JS | JNS | JP | JNP | JL | JNL | JLE | JNLE |
| 8 | Immediate Grpl | | | Grpl | TEST | | XCNG | | MOV | | | | MOV | LEA | MOV | POP |
| | Eb,Ib | Ev,Iv | | Ev,Iv | Eb,Gb | Ev,Gv | Eb,Gb | Ev,Gv | Eb,Gb | Ev,Gv | Gb,Eb | Gv,Ev | Ew,Sw | Gv,M | Sw,Ew | Ev |
| 9 | NOP | XCHG word or double-word register with eAX | | | | | | | CBW | CWD | CALL Ap | WAIT | PUSHF Fv | POPF Fv | SAHF | LAHF |
| | | eCX | eDX | eBX | eSP | eBP | eSI | eDI | | | | | | | | |
| A | MOV | | | | MOVSB | MOVSW/D | CMPSB | CMPSW/D | TEST | | STOSB | STOSW/D | LODSB | LODSW/D | SCASB | SCASW/D |
| | AL,Ob | eAX,Ov | Ob,AL | Ov,eAX | Xb,Yb | Xv,Yv | Xb,Yb | Xv,Yv | AL,Ib | eAX,Iv | Yb,AL | Yv,eAX | AL,Xb | eAX,Xv | AL,Xb | eAX,Xv |
| B | MOV immediate byte into byte register | | | | | | | | MOV immediate word or double into word or double register | | | | | | | |
| | AL | CL | DL | BL | AH | CH | DH | BH | eAX | eCX | eDX | eBX | eSP | eBP | eSI | eDI |
| C | Shift Grp2 | | RET near | | LES | LDS | MOV | | ENTER | LEAVE | RET far | | INT | INT | INTO | IRET |
| | Eb,Ib | Ev,Iv | Iw | | Gv,Mp | Gv,Mp | Eb,Ib | Ev,Iv | Iw,Ib | | Iw | | 3 | Ib | | |
| D | Shift Grp2 | | | | AAM | AAD | | XLAT | ESC(Escape to coprocessor instruction set) | | | | | | | |
| | Eb,1 | Ev,1 | Eb,CL | Ev,CL | | | | | | | | | | | | |
| E | LOOPNE | LOOPE | LOOP | JCXZ | IN | | OUT | | CALL | JNP | | | IN | | OUT | |
| | Jb | Jb | Jb | Jb | AL,Ib | eAX,Ib | Ib,AL | Ib,eAX | Av | Jv | Ap | Jb | AL,DX | eAX,DX | DX,AL | DX,eAX |
| F | LOCK | | REPNE | REP REPE | HLT | CMC | Unary Grp3 | | CLC | STC | CLI | STI | CLD | STD | INC/DEC | Indirct |
| | | | | | | | Eb | Ev | | | | | | | Grp4 | Grp5 |

Two-Byte Opcode Map (first byte is 0FH)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Grp6 | Grp7 | LAR Gw,Ew | LSL Gv,Ew | | | CLTS | | | | | | | | | |
| 1 | | | | | | | | | | | | | | | | |
| 2 | MOV Cd,Rd | MOV Dd,Rd | MOV Rd,Cd | MOV Rd,Dd | MOV Td,Rd | | MOV Rd,Td | | | | | | | | | |
| 3 | | | | | | | | | | | | | | | | |
| 4 | | | | | | | | | | | | | | | | |
| 5 | | | | | | | | | | | | | | | | |
| 6 | | | | | | | | | | | | | | | | |
| 7 | | | | | | | | | | | | | | | | |
| 8 | Long-displacement jump on condition (Jv) | | | | | | | | Long-displacement jump on condition (Jv) | | | | | | | |
| | JO | JNO | JB | JNB | JZ | JNZ | JBE | JNBE | JS | JNS | JP | JNP | JL | JNL | JLE | JNLE |
| 9 | Byte Set on condition (Eb) | | | | | | | | SETS | SETNS | SETP | SETNP | SETL | SETNL | SETLE | SETNLE |
| | SETO | SETNO | SETB | SETNB | SETZ | SETNZ | SETBE | SETNBE | | | | | | | | |
| A | PUSH FS | POP FS | | BT Ev,Gv | SHLD EvGvIb | SHLD EvGvCL | | | PUSH GS | POP GS | | BTS Ev,Gv | SHRD EvGvIb | SHRD EvGvCL | | IMUL Gv,Ev |
| B | | | LSS Mp | BTR Ev,Gv | LFS Mp | LGS Mp | MOVZX Gv,Eb | MOVZX Gv,Ew | | | Grp-8 Ev,Ib | BTC Ev,Gv | BSF Gv,Ev | BSR Gv,Ev | MOVSX Gv,Eb | MOVSX Gv,Ew |
| C | | | | | | | | | | | | | | | | |
| D | | | | | | | | | | | | | | | | |
| E | | | | | | | | | | | | | | | | |
| F | | | | | | | | | | | | | | | | |

Opcodes determined by bits 5,4,3 of modR/M byte:

| Group | mod nnn R/M 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 1 | ADD | OR | ADC | SBB | AND | SUB | XOR | CMP |
| 2 | ROL | ROR | RCL | RCR | SHL | SHR | | SAR |
| 3 | TEST Ib/Iv | | NOT | NEG | MUL AL/eAX | IMUL AL/eAX | DIV AL/eAX | IDIV AL/eAX |
| 4 | INC Eb | DEC Eb | | | | | | |
| 5 | INC Ev | DEC Ev | CALL Ev | CALL eP | JMP Ev | JMP Ep | PUSH Ev | |

Opcodes determined by bits 5,4,3 of modR/M byte:

| Group | mod nnn R/M 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 6 | SLDT Ew | STR Ew | LLDT Ew | LTR Ew | VERR Ew | VERW Ew | | |
| 7 | SGDT Ms | SIDT Ms | LGDT Ms | LIDT Ms | SMSW Ew | | LMSW Ew | |
| 8 | | | | | BT | BTS | BTR | BTC |

# Appendix B  Complete Flag Cross-Reference

---

Key to Codes

T     = instruction tests flag

M     = instruction modifies flag
        (either sets or resets depending on operands)

0     = instruction resets flag

1     = instruction sets flag

——    = instruction's effect on flag is undefined

R     = instruction restores prior value of flag

blank = instruction does not affect flag

| Instruction | OF | SF | ZF | AF | PF | CF | TF | IF | DF | NT | RF |
|---|---|---|---|---|---|---|---|---|---|---|---|
| AAA | —— | —— | —— | TM | —— | M | | | | | |
| AAD | —— | M | M | —— | M | —— | | | | | |
| AAM | —— | M | M | —— | M | —— | | | | | |
| AAS | —— | —— | —— | TM | —— | M | | | | | |
| ADC | M | M | M | M | M | TM | | | | | |
| ADD | M | M | M | M | M | M | | | | | |
| AND | 0 | M | M | —— | M | 0 | | | | | |
| ARPL | | | M | | | | | | | | |
| BOUND | | | | | | | | | | | |
| BSF/BSR | —— | —— | M | —— | —— | —— | | | | | |
| BT/BTS/BTR/BTC | —— | —— | —— | —— | —— | M | | | | | |
| CALL | | | | | | | | | | | |
| CBW | | | | | | | | | | | |
| CLC | | | | | | 0 | | | | | |
| CLD | | | | | | | | | 0 | | |
| CLI | | | | | | | | 0 | | | |
| CLTS | | | | | | | | | | | |
| CMC | | | | | | M | | | | | |
| CMP | M | M | M | M | M | M | | | | | |
| CMPS | M | M | M | M | M | M | | | T | | |
| CWD | | | | | | | | | | | |
| DAA | —— | M | M | TM | M | TM | | | | | |
| DAS | —— | M | M | TM | M | TM | | | | | |
| DEC | M | M | M | M | M | | | | | | |
| DIV | —— | —— | —— | —— | —— | —— | | | | | |
| ENTER | | | | | | | | | | | |
| ESC | | | | | | | | | | | |
| HLT | | | | | | | | | | | |
| IDIV | —— | —— | —— | —— | —— | —— | | | | | |
| IMUL | M | —— | —— | —— | —— | M | | | | | |
| IN | | | | | | | | | | | |
| INC | M | M | M | M | M | | | | | | |
| INS | | | | | | | | | T | | |
| INT | | | | | | | 0 | | | 0 | |

| Instruction | OF | SF | ZF | AF | PF | CF | TF | IF | DF | NT | RF |
|---|---|---|---|---|---|---|---|---|---|---|---|
| INTO | T | | | | | | 0 | | | 0 | |
| IRET | R | R | R | R | R | R | R | R | R | T | |
| Jcond | T | T | T | T | T | | | | | | |
| JCXZ | | | | | | | | | | | |
| JMP | | | | | | | | | | | |
| LAHF | | | | | | | | | | | |
| LAR | | | M | | | | | | | | |
| LDS/LES/LSS/LFS/LGS | | | | | | | | | | | |
| LEA | | | | | | | | | | | |
| LEAVE | | | | | | | | | | | |
| LGDT/LIDT/LLDT/LMSW | | | | | | | | | | | |
| LOCK | | | | | | | | | | | |
| LODS | | | | | | | | | T | | |
| LOOP | | | | | | | | | | | |
| LOOPE/LOOPNE | | | T | | | | | | | | |
| LSL | | | M | | | | | | | | |
| LTR | | | | | | | | | | | |
| MOV | | | | | | | | | | | |
| MOV control, debug | — | — | — | — | — | — | | | | | |
| MOVS | | | | | | | | | T | | |
| MOVSX/MOVZX | | | | | | | | | | | |
| MUL | M | — | — | — | — | M | | | | | |
| NEG | M | M | M | M | M | M | | | | | |
| NOP | | | | | | | | | | | |
| NOT | | | | | | | | | | | |
| OR | 0 | M | M | — | M | 0 | | | | | |
| OUT | | | | | | | | | | | |
| OUTS | | | | | | | | | T | | |
| POP/POPA | | | | | | | | | | | |
| POPF | R | R | R | R | R | R | R | R | R | R | |
| PUSH/PUSHA/PUSHF | | | | | | | | | | | |
| RCL/RCR 1 | M | | | | | TM | | | | | |
| RCL/RCR count | — | | | | | TM | | | | | |
| REP/REPE/REPNE | | | | | | | | | | | |
| RET | | | | | | | | | | | |
| ROL/ROR 1 | M | | | | | M | | | | | |
| ROL/ROR count | — | | | | | M | | | | | |
| SAHF | | R | R | R | R | R | | | | | |
| SAL/SAR/SHL/SHR 1 | M | M | M | — | M | M | | | | | |
| SAL/SAR/SHL/SHR count | — | M | M | — | M | M | | | | | |
| SBB | M | M | M | M | M | TM | | | | | |
| SCAS | M | M | M | M | M | M | | | T | | |
| SET cond | T | T | T | | T | T | | | | | |
| SGDT/SIDT/SLDT/SMSW | | | | | | | | | | | |
| SHLD/SHRD | — | M | M | — | M | M | | | | | |
| STC | | | | | | 1 | | | | | |
| STD | | | | | | | | | 1 | | |
| STI | | | | | | | | 1 | | | |
| STOS | | | | | | | | | T | | |
| STR | | | | | | | | | | | |
| SUB | M | M | M | M | M | M | | | | | |
| TEST | 0 | M | M | — | M | 0 | | | | | |
| VERR/VERRW | | | M | | | | | | | | |
| WAIT | | | | | | | | | | | |
| XCHG | | | | | | | | | | | |
| XLAT | | | | | | | | | | | |
| XOR | 0 | M | M | — | M | 0 | | | | | |

# Appendix C Status Flag Summary

---

Status Flags' Functions

| Bit | Name | Function |
|---|---|---|
| 0 | CF | Carry Flag —— Set on high-order bit carry or borrow; cleared otherwise. |
| 2 | PF | Parity Flag —— Set if low-order eight bits of result contain an even number of 1 bits; cleared otherwise. |
| 4 | AF | Adjust flag —— Set on carry from or borrow to the low order four bits of AL; cleared otherwise. Used for decimal arithmetic. |
| 6 | ZF | Zero Flag —— Set if result is zero; cleared otherwise. |
| 7 | SF | Sign Flag —— Set equal to high-order bit of result (0 is positive, 1 if negative). |
| 11 | OF | Overflow Flag —— Set if result is too large a positive number or too small a negative number (excluding sign-bit) to fit in destination operand; cleared otherwise. |

Key to Codes

```
T     = instruction tests flag
M     = instruction modifies flag
        (either sets or resets depending on operands)
0     = instruction resets flag
——    = instruction's effect on flag is undefined
blank = instruction does not affect flag
```

| Instruction | OF | SF | ZF | AF | PF | CF |
|---|---|---|---|---|---|---|
| AAA | — | — | — | TM | — | M |
| AAS | — | — | — | TM | — | M |
| AAD | — | M | M | — | M | — |
| AAM | — | M | M | — | M | — |
| DAA | — | M | M | TM | M | TM |
| DAS | — | M | M | TM | M | TM |
| ADC | M | M | M | M | M | TM |
| ADD | M | M | M | M | M | M |
| SBB | M | M | M | M | M | TM |
| SUB | M | M | M | M | M | M |
| CMP | M | M | M | M | M | M |
| CMPS | M | M | M | M | M | M |
| SCAS | M | M | M | M | M | M |
| NEG | M | M | M | M | M | M |
| DEC | M | M | M | M | M | |
| INC | M | M | M | M | M | |
| IMUL | M | — | — | — | — | M |
| MUL | M | — | — | — | — | M |
| RCL/RCR 1 | M | | | | | TM |
| RCL/RCR count | — | | | | | TM |
| ROL/ROR 1 | M | | | | | M |
| ROL/ROR count | — | | | | | M |
| SAL/SAR/SHL/SHR 1 | M | M | M | — | M | M |
| SAL/SAR/SHL/SHR count | — | M | M | — | M | M |
| SHLD/SHRD | — | M | M | — | M | M |
| BSF/BSR | — | — | M | — | — | — |
| BT/BTS/BTR/BTC | — | — | — | — | — | M |
| AND | 0 | M | M | — | M | 0 |
| OR | 0 | M | M | — | M | 0 |
| TEST | 0 | M | M | — | M | 0 |
| XOR | 0 | M | M | — | M | 0 |

# Appendix D Condition Codes

---

Note:
The terms "above" and "below" refer to the relation between two unsigned values (neither SF nor OF is tested). The terms "greater" and "less" refer to the relation between two signed values (SF and OF are tested).

---

Definition of Conditions

(For conditional instructions Jcond, and SETcond)

| Mnemonic | Meaning | Instruction Subcode | Condition Tested |
|---|---|---|---|
| O | Overflow | 0000 | OF = 1 |
| NO | No overflow | 0001 | OF = 0 |
| B<br>NAE | Below<br>Neither above nor equal | 0010 | CF = 1 |
| NB<br>AE | Not below<br>Above or equal | 0011 | CF = 0 |
| E<br>Z | Equal<br>Zero | 0100 | ZF = 1 |
| NE<br>NZ | Not equal<br>Not zero | 0101 | ZF = 0 |
| BE<br>NA | Below or equal<br>Not above | 0110 | (CF or ZF) = 1 |
| NBE<br>NA | Neither below nor equal<br>Above | 0111 | (CF or ZF) = 0 |
| S | Sign | 1000 | SF = 1 |
| NS | No sign | 1001 | SF = 0 |
| P<br>PE | Parity<br>Parity even | 1010 | PF = 1 |
| NP<br>PO | No parity<br>Parity odd | 1011 | PF = 0 |
| L<br>NGE | Less<br>Neither greater nor equal | 1100 | (SF xor OF) = 1 |
| NL<br>GE | Not less<br>Greater or equal | 1101 | (SF xor OF) = 0 |
| LE<br>NG | Less or equal<br>Not greater | 1110 | ((SF xor OF) or ZF) = 1 |
| NLE<br>G | Neither less nor equal<br>Greater | 1111 | ((SF xor OF) or ZF) = 0 |