Advanced Micro Devices

80C286/AMD 80C287™ Programmer's Reference Manual

14554A

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

ORGANIZATION OF THIS MANUAL

This chapter includes a general introduction to the 80C286 features and operation. Chapters 2–11 serve as an introduction to the 80C286 architecture and as a reference guide. Chapters 2–4 should be read to gain a basic familiarity with the 80C286 architecture. They provide detailed information on memory segmentation, registers, addressing modes and the general (application level) 80C286 instruction set.

Chapters 6–11 are aimed primarily at system programmers and discuss the more advanced architectural features of the 80C286 that are available when the processor is in protected mode. This part provides details on memory management, protection mechanisms, and task switching.

The last four chapters of this manual describe the AMD 80C287™ math coprocessor from a software design perspective. Applications and systems software concepts are addressed.

A Glossary following Chapter 15 defines terminology related to the AMD 80C287 math coprocessor and floating-point calculations.

Appendices at the end of this document provide related information to both the 80C286 CPU and the AMD 80C287 math coprocessor.

CHAPTER DESCRIPTIONS

Chapter 2 discusses specific features of the 80C286 architecture significant for application programmers. The information can also function as an introduction to the machine for system programmers. The chapter discusses memory organization and segmentation, processor registers, addressing modes, and instruction formats.

Chapter 3 explains the core instructions for the 8086 family.

Chapter 4 presents the extended instructions both the 80186 and 80C286 processors share.

Chapter 5 describes the system programmer's view of the 80C286 when operated in real address mode.

Chapter 6 describes the 80C286 address translation mechanisms supporting virtual memory. It discusses segment descriptors, global and local descriptor tables, and descriptor caches.

Chapter 7 describes the 80C286 protection features, including privilege levels, segment attributes, access restrictions, and call gates.

Chapter 8 describes the 80C286 mechanisms supporting concurrent tasks: context-switching, task state segments, task gates, and interrupt tasks.

Chapter 9 explains interrupt and trap handling, paying special attention to the exception traps, or faults, which may occur in protected mode. The chapter also discusses interrupt gates, trap gates, and the interrupt descriptor table.

Chapter 10 details the actual instructions used to implement the 80C286 memory management, protection, and task support features. System registers, privileged instructions, and the initial machine state are explained.

Chapter 11 describes several advanced topics, including special segment attributes and pointer validation.

Chapter 12 is an overview of the AMD 80C287 math coprocessor and a review of its use in numeric computation.

Chapter 13 gives software designers detailed information for generating applications for using the AMD 80C287 math coprocessor in 80C286-based systems.

Chapter 14 provides details of AMD 80C287 architecture and operational characteristics as well as other information of interest to system-software developers.

Chapter 15 contains assembly-language programming examples for the AMD 80C287 math coprocessor. Included are examples of conditional branching, converting floating-point values to ASCII representations, and trigonometric function calculation.

INTRODUCTION TO THE 80C286

Of the 8086 series of microprocessors, which includes the 8086, the 8088, the 80186, the 80188, and the 80C286, the 80C286 is the most powerful. It is designed for applications requiring very high performance and is an excellent choice for sophisticated high-end applications that benefit from its advanced architectural features: memory management, protection mechanisms, task management, and virtual memory support. On a single VLSI chip, the 80C286 provides computational and architectural characteristics normally associated with much larger minicomputers.

This rest of this chapter provides an overview of the 80C286 architecture. The 80C286 architecture extends the 8086 architecture, so some of this overview material may be new and unfamiliar to previous users of the 8086 and similar microprocessors. But because the 80C286 is an evolutionary development, the new architecture is superimposed upon the industry standard 8086 in such a way as to affect only the design and programming of operating systems and other system software.

GENERAL ATTRIBUTES

The base architecture of the 80C286 has many features in common with the architecture of other members of the 8086 family, such as byte-addressable memory, I/O interfacing hardware, interrupt vectoring, and support for both multiprocessing and processor extensions. A set of addressing modes and basic instructions are common to the entire family. In addition, the 80C286 base architecture includes a number of extensions that add to the versatility of the computer.

There are two modes of operation in which the 80C286 processor can function. In one mode only the base architecture is available to programmers; in the other mode a number of very powerful advanced features have been added, including support for virtual memory, multi-tasking, and a sophisticated protection mechanism.

The base architecture of the 80C286 was designed to support high-level programming in such languages as Pascal, C, or PL/M. The register set and instructions are

particularly well-suited to compiler-generated code. The addressing modes described in Chapter 2 allow efficient addressing of complex data structures, such as static and dynamic arrays, records,and arrays within records, which are commonly supported by high-level languages. Along with bytes and words, the data types supported by the architecture include high-level language constructs such as strings, BCD, and floating point.

The 80C286 memory architecture was designed to support modular programming techniques. Memory is divided into segments that can be used to contain procedures and data structures. Each segment may be of arbitrary size. Segmentation provides several advantages over more conventional linear memory architectures. Because segments can contain meaningful program units and data and more compact code, segmentation supports structured software. References within a segment can be shorter (and locality of references usually insures that the next few references fall within the same segment). Segmentation also provides for efficient implementation of sophisticated memory management, virtual memory, and memory protection.

New instructions have also been added to the base architecture to give hardware support for procedure invocations, parameter passing, and array bounds checking.

MODES OF OPERATION

The 80C286 can be operated in two different modes: real address mode or protected virtual address mode (sometimes referred to as protected mode). In either mode of operation, the 80C286 provides an upwardly compatible addition to the 8086 family of processors.

In real address mode, the 80C286 operates much like a very high-performance 8086. With few exceptions, programs written for the 8086 or the 80186 can be executed in this mode without any modification (exceptions are described in Appendix C). Such upward compatibility extends even to the object code level; an 8086 program stored in read-only memory, for example, will execute successfully in 80C286 real address mode. A number of instructions not found on the 8086 are provided by the 80C286 operating in real address mode. These additional instructions (also present with the 80186) allow for efficient subroutine linkage, parameter validation, index calculations, and block I/O transfers.

The advanced architectural features and full capabilities of the 80C286 are realized in the native protected mode. These advanced features include sophisticated mechanisms to support data protection, system integrity, task concurrency, and memory management, including virtual storage. However, even in protected mode, the 80C286 remains upwardly compatible with most 8086 and 80186 application programs. Most 8086 applications programs can be recompiled or reassembled on the 80C286 in this mode.

ADVANCED FEATURES

The architectural features described in this chapter are common to both operating modes of the 80C286 processor. Protected mode provides a number of advanced features in addition to these common features, including a greatly extended physical and logical address space, new instructions, and support for additional hardware-recognized data structures. The protected mode of the 80C286 also includes a sophisticated memory management and multilevel protection mechanism. Full hardware support allows for multi-tasking and task-switching operations.

Memory Management

The memory architecture of the 80C286 in protected mode represents a significant advance over that of the 8086. The physical address space has expanded from 1 megabyte to 16 Mb (2^{24} bytes), and the virtual address space (the address space visible to a program) has increased from 1 Mb to 1 Gb (2^{30} bytes). In addition, separate virtual address spaces are provided for each task in a multi-tasking system.

The 80C286 supports on-chip memory management rather than relying on an external memory management unit. Advantages of the one-chip solution are that no software is required to manage an external memory management unit, performance is much better, and hardware designs are significantly simpler.

The 80C286 architecture includes mechanisms to allow for efficient implementation of virtual memory systems. (In virtual memory systems, the user treats the combination of main and external storage as a single large memory. The user can write large programs without concern about physical memory limitations of the system because the operating system places some of the user programs and data in external storage and brings them into main memory only as they are needed.) All instructions that might cause a segment-not-present fault are fully restartable. Because of this, a not-present segment can be loaded from external storage, and the task can be restarted from where the fault occurred.

Like all members of the 8086 series, the 80C286 supports a segmented memory architecture. It also fully integrates memory segmentation into a comprehensive protection scheme. To protect segments from inadvertent misuse, this scheme includes hardware-enforced length and type checking.

Task Management

The 80C286 design supports multi-tasking systems with direct support for the concept of a task. For example, task state segments are hardware-recognized and hardware-manipulated structures that contain information on the current state of all tasks in the system.

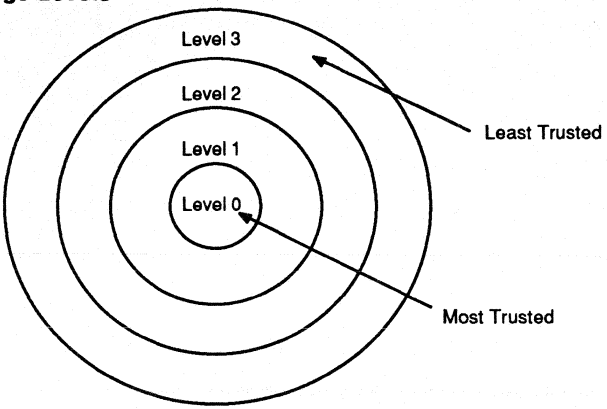
Context switching (task switching) can be efficiently invoked with a single instruction. Separate logical address spaces are reserved for each task in the system. Other mechanisms support inter-task communication, synchronization, memory sharing, and task scheduling. Chapter 8 describes Task management.

Protection Mechanisms

The 80C286 allows the system designer to define a comprehensive protection policy to be uniformly and continuously applied to all ongoing system operations. Such policies may ensure system reliability, privacy of data, rapid error recovery, and separation of multiple users.

The protection mechanisms of the 80C286 are based on a hierarchy of trust. The four privilege levels range from Level 0 (most trusted) to Level 3 (least trusted). As a rule, Level 0 is reserved for the operating system kernel. The privilege levels may be viewed as concentric rings, the most privileged level being in the center (see Figure 1-1).

Figure 1-1 **Four Privilege Levels**



This four-level scheme provides system reliability, flexibility, and design options unavailable with the typical two-level (supervisor/user) separation other processors provide. A four-level division can separate kernel, executive, system services, and application software, assigning different privileges to each level.

At any one time, a task may be executed at one of the four levels. All data segments and code segments are also assigned to privilege levels. Therefore, a task executing at one level cannot access data from a more privileged level, nor can it call a procedure at a less privileged level (i.e., a less privileged procedure may not be trusted to do work for it). Because of this, both access to data and transfer of control are appropriately restricted.

The logical address spaces local to different tasks can be completely separated, and thereby provide users with automatic protection against accidental or malicious interference by others. The hardware also provides immediate detection of a variety of fault and error conditions, which can be useful in software development and maintenance.

Finally, because they are integrated into the memory management and protection hardware of the processor itself, these protection mechanisms require relatively little system overhead.

Support for Operating Systems

Most operating systems involve some degree of concurrency, for multiple tasks vying for system resources. The task management mechanisms previously described provide the 80C286 with inherent support for multi-tasking systems. The advanced memory management features of the 80C286 also allow implementation of sophisticated virtual memory systems.

Operating system implementors have found that a multilevel approach to system services provides for better security and more reliable systems. For instance, a very secure kernel might implement critical functions such as task scheduling and resource allocation, while less fundamental functions (such as I/O) are built around the kernel. Such a layered approach makes program development and enhancement simpler and facilitates error detection and debugging. The four-level privilege scheme of the 80C286 supports this layered approach.

CHAPTER 2

80C286 BASE ARCHITECTURE

This chapter discusses the 80C286 application programming environment as seen by assembly language programmers and introduces programmers to those features of the 80C286 architecture that directly affect the design and implementation of 80C286 application programs.

MEMORY ORGANIZATION AND SEGMENTATION

The main memory of the 80C286 system makes up its physical address space. The address space is organized as a sequence of 8-bit quantities (bytes). Each byte is assigned a unique address ranging from a minimum of 0 to a maximum of 2^{20} (1 Mb) in real address mode, and a maximum of 2^{24} (16 Mb) in protected mode.

A virtual address space is defined as the organization of memory as viewed by a program. Virtual address space may be organized in units of bytes. As with the 8086 itself, programs view physical memory directly in the real address mode by manipulating pure physical addresses. Therefore, the virtual address space is identical to the physical address space (1 Mb).

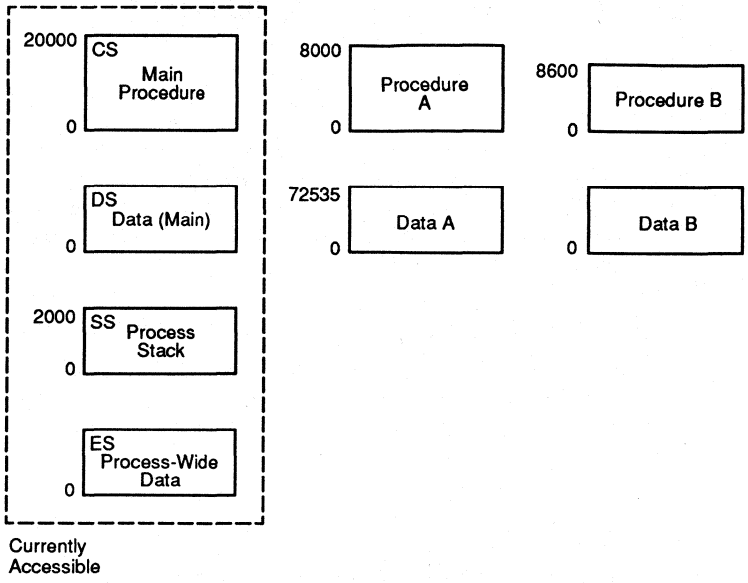
However, protected mode programs have no direct access to physical addresses. Rather, memory is viewed as a much larger virtual address space of 2^{30} bytes (1 Gb), which is mapped onto the 16-Mb physical address space by address translation mechanisms described in Chapter 6.

The programmer views the virtual address space on the 80C286 as a collection of up to sixteen thousand linear subspaces, each having a specified size or length. Each of these linear address spaces is a segment (a logical unit of contiguous memory). Segment sizes range from one byte up to 64K (65,536) bytes.

Memory segmentation of the 80C286 supports the logical structure of programs and data in memory. Programs are written as modules of code and data rather than as single linear sequences of instructions and data. For example, a main routine and several separate procedures may be included in a program code. Data may be organized into various data structures, some remaining private and some being shared with other programs in the system. Run-time stacks constitute an additional data requirement. Each module of code and data may vary in size or vary dynamically with program execution.

Segmentation supports this logical structure (see Figure 2-1). Each meaningful program module may be separately contained in individual segments. Of course, the degree of modularization depends on the requirements of a particular application. Segmentation benefits almost all applications because programs execute faster and require less space. It also simplifies the design of structured software.

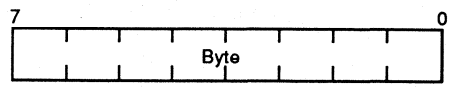
Figure 2-1 **Segmented Virtual Memory**



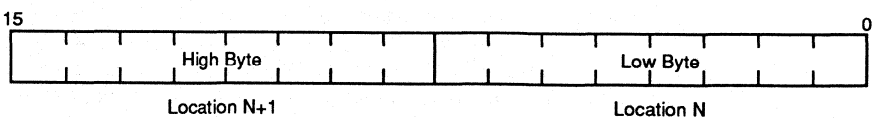
DATA TYPES

The fundamental units in which the 80C286 manipulates data are bytes and words.

A byte consists of 8 contiguous bits starting on an addressable byte boundary. Starting from the right, the bits are numbered 0 through 7, the most significant being bit 7:

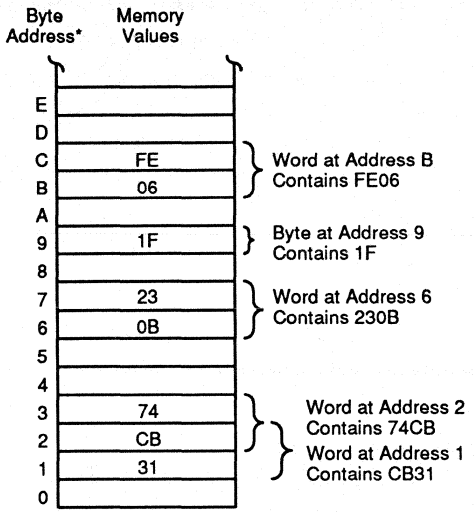


A word is defined as two contiguous bytes starting on an arbitrary byte boundary; therefore, a word contains 16 bits. The bits of a word are numbered 0 through 15, starting at the right. The most significant bit in a word is bit 15. The byte containing bit 0 of the word is called the low byte, and the byte containing bit 15 is called the high byte.



Each byte within a word has its own particular address, and the address of the word is the smaller of the two addresses. The byte at the lower address contains the eight least significant bits of the word; the byte at the higher address contains the eight most significant bits. Figure 2-2 illustrates the arrangement of bytes within words.

Figure 2-2 Bytes and Words in Memory



*Note: All values in hexadecimal.

14554A-003

Note: A word need not be aligned at an even-numbered byte address. This provides maximum flexibility in data structures (e.g., records containing mixed byte and word entries) and efficient utilization of memory. Actual transfers of data between the processor and memory take place at physically aligned word boundaries, but the 80C286 converts requests for unaligned words into the appropriate sequences of requests acceptable to the memory interface. Such odd aligned word transfers, however, may affect performance by requiring two memory cycles rather than one to transfer the word. Therefore, data structures should be designed whenever possible in such a way that word operands are aligned on word boundaries for maximum system performance. Due to instruction prefetching and queueing with the CPU, instructions are not required to be aligned on word boundaries, and no performance is lost if they are not.

Although the fundamental data types of operands are bytes and words, the processor also supports additional interpretations on these bytes or words. Depending on the instruction referencing the operand, these following additional data types can be recognized:

Integer:	A signed binary numeric value within an 8-bit byte or a 16-bit word. A 2's complement representation is assumed by all operations. (Signed 32- and 64-bit integers are supported by the AMD 80C287 math coprocessor.)
Ordinal:	An unsigned binary numeric value within an 8-bit byte or 16-bit word.
Pointer:	A 32-bit address quantity consisting of a segment selector component and an offset component. Each component consists of a 16-bit word.
String:	A contiguous sequence of bytes or words containing from 1 byte to 64K bytes.

ASCII: A byte representation of alphanumeric and control characters using the standard ASCII character representation.

BCD: A byte (unpacked) representation of decimal digits (0–9).

Packed BCD: A byte (packed) representation of two decimal digits (0–9). Each nibble of the byte stores one digit.

Floating Point: A signed 32-, 64-, or 80-bit representation of a real number. (Floating operands are supported by the AMD 80C287 math coprocessor.)

Figure 2-3 is a graphic representation of the data types supported by the 80C286. 80C286 arithmetic operations may be performed on five types of numbers:

1. Unsigned binary
2. Signed binary (integers)
3. Unsigned packed decimal
4. Unsigned unpacked decimal
5. Floating point

Binary numbers are between 8 and 16 bits long. Decimal numbers are stored in bytes; there are two digits per byte for packed decimal and one digit per byte for unpacked decimal. The processor always assumes that the operands specified in arithmetic instructions contain data representing valid numbers for the type of instruction being performed. Invalid data may lead to unpredictable results.

Unsigned binary numbers may be 8 or 16 bits long; all bits are considered when determining a number's magnitude. The value of an 8-bit unsigned binary number ranges from 0 to 255; 16 bits may represent values from 0 through 65,535. Addition, subtraction, multiplication and division operations are also available for unsigned binary numbers.

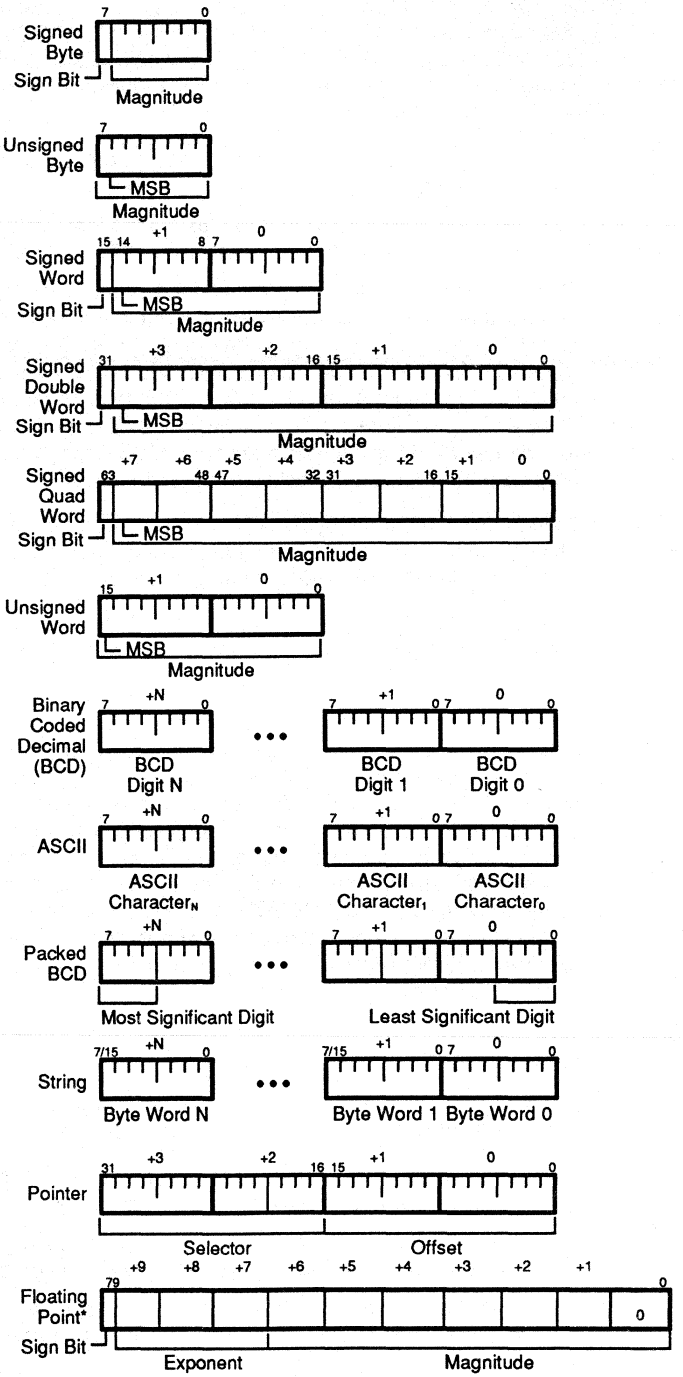
Signed binary numbers (integers) may be either 8 or 16 bits in length. The high-order (leftmost) bit is interpreted as the number's sign: 0 being positive and 1 being negative. Negative numbers are represented by standard two's complement notation. Since the high-order bit is used for a sign, an 8-bit integer ranges from –128 through +127, and 16-bit integers range from –32,768 through +32,767. The value zero uses a positive sign.

Separate multiplication and division operations are provided for both signed and unsigned binary numbers, but the same addition and subtraction instructions are used. In order to detect overflow into the sign bit, conditional jump instructions and an Interrupt on Overflow instruction can be used after an unsigned operation on an integer.

Unpacked decimal numbers are stored as unsigned byte quantities, and a single digit is stored in each byte. The low-order half-byte determines the magnitude of the number; valid hexadecimal values, 0–9, are interpreted as decimal numbers. For multiplication and division, the high-order half-byte must be zero, and it may contain any value for addition and subtraction.

Arithmetic on unpacked decimal numbers is done in two steps. First, the use of unsigned binary addition, subtraction, and multiplication operations produces an intermediate result. Then an adjustment instruction changes the value to a final correct unpacked decimal number. The division operation is performed similarly. However,

Figure 2-3 80C286 and AMD 80C287 Math Coprocessor Supported Data Types



*Supported by 80C286 using the AMD 80C287 math coprocessor.

09729B-006

the adjustment is carried out on the two digit numerator operand in register AX first. An unsigned binary division instruction that produces a correct result follows.

Although the unpacked decimal numbers and the ASCII character representations of the digits 0–9 are similar, the high-order half-byte of an ASCII numeral is always 3. Under the following conditions, unpacked decimal arithmetic may be performed on ASCII numeric characters:

- Before multiplication or division, the high-order half-byte of an ASCII numeral must be set to 0H.
- Unpacked decimal arithmetic leaves the high-order half-byte set to 0H, and it must be set to 3 to produce a valid ASCII numeral.

Packed decimal numbers are maintained as unsigned byte quantities. The byte is regarded as having one decimal digit in each half-byte (nibble), the most significant being the digit in the high-order half-byte. Values 0–9 are valid in each half-byte, and a packed decimal number ranges from 0 to 99. Additions and subtractions are completed in two steps. After an addition or subtraction instruction is performed to produce an intermediate result, an adjustment operation changes the intermediate value to a final correct packed decimal result. Multiplication and division adjustments can only be performed on unpacked decimal numbers.

Strings are contiguous bytes or words that range from 1–64K bytes in length. They usually contain ASCII or other character data representations. String manipulation instructions on the 80C286 can move, examine, or modify a string.

If the AMD 80C287 coprocessor is present in the system, the 80C286 architecture can also support floating point numbers, 32- and 64-bit integers, and 18-digit BCD data types.

Real numbers are supported and stored in a three-field binary format by the AMD 80C287 math coprocessor, as required by IEEE standard 754 for floating-point numerics (see Figure 2-3). While the number's significant digits are kept in the significand field, the exponent field locates the binary point within the significant digits, and thus determines the number's magnitude. The sign field shows whether the number is positive or negative. (The exponent and significand are analogous to the terms characteristic and mantissa, which are used to describe floating-point numbers on some computers.) The AMD 80C287 math coprocessor uses this format with various length significands and exponents to support single precision, double precision and extended (80-bit) precision floating point data types. Negative numbers and positive numbers differ only in their sign bits.

REGISTERS

The 80C286 has fourteen registers that are of interest to the application programmer. These registers may be grouped into four basic categories, as shown in Figure 2-4:

- General registers. These eight 16-bit general-purpose registers primarily store operands for arithmetic and logical operations.
- Segment registers. These four special-purpose registers determine which segments of memory may be addressed at any given time.
- Status and Control registers. These three special-purpose registers record and alter specific aspects of the 80C286 processor state.

General Registers

The 16-bit registers AX, BX, CX, DX, SP, BP, SI, and DI are the general registers of the 80C286. They are used interchangeably to hold the operands of logical and arithmetic operations.

However, some instructions and addressing modes delegate certain general registers to specific uses. BX and BP often contain the base address of data structures in memory (for instance, the starting address of an array); because of this, they are often referred to as the base registers. Likewise, SI and DI are often used to store an index value that will be incremented to step through a data structure; thus, they are called the index registers. Registers SP and BP are dedicated to stack manipulation. Although SP and BP typically contain offsets into the current stack, SP generally contains the offset of the top of the stack, while BP contains the offset or base address of the current stack frame. Register usage for individual instructions is explained in chapters 3 and 4.

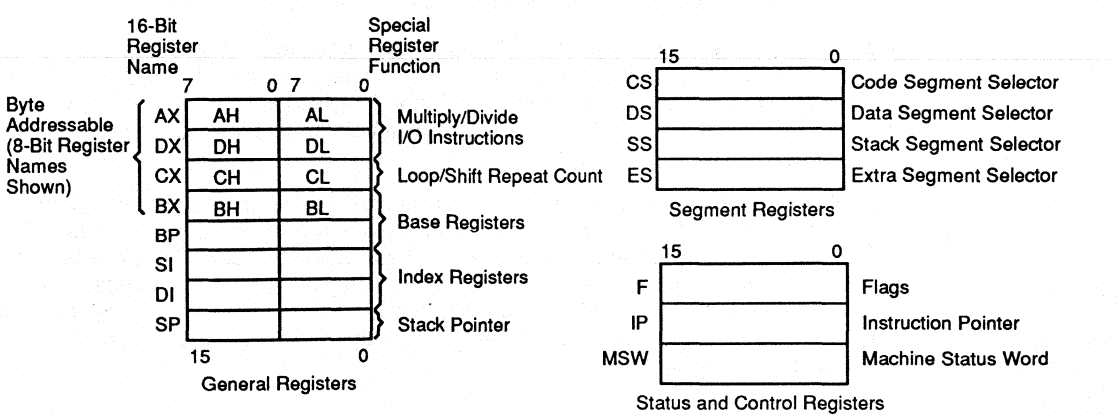
Figure 2-4 shows how eight byte registers overlap four of the 16-bit general registers. These registers are called AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes). They overlap AX, BX, CX, and DX and can be used either in their entirety or as individual 8-bit registers. The handling of both 8- and 16-bit data elements is simplified by this dual interpretation.

Memory Segmentation and Segment Registers

Many different code modules (or segments) and different types of data segments comprise complete programs. However, only a small subset of a program's segments are actually in use at any given time during program execution. This subset generally includes code, data, and possibly a stack. To take advantage of this, the 80C286 architecture provides mechanisms to support direct access to the working set of a program's execution environment and access on demand to additional segments.

Four segments of memory are immediately accessible to an executing 80C286 program at any time. The DS, ES, SS, and CS segment registers identify these four current segments, and each register specifies a particular kind of segment, characterized by the associated mnemonics (code, stack, data, or extra) shown in Figure 2-4.

Figure 2-4 80C286 Base Architecture Register Set



An executing program is given concurrent access to four individual segments of memory—a code segment, a stack segment, and two data segments—by the four segment registers. Each register chooses a segment, uniquely determining the one particular segment from the numerous segments in memory. For this reason, the 16-bit contents of a segment register is called a segment selector.

Once a segment is chosen, a base address is associated with it. A 16-bit offset from the segment's base address must be supplied in order to address an element within a segment. Together, the 16-bit segment selector and the 16-bit offset form the high and low order halves, respectively, of a 32-bit virtual address pointer. Usually only the lower 16 bits of the pointer, called the offset, need to be specified by an instruction once a segment is selected. When only a 16-bit offset is specified, simple rules denote which segment register is used to form an address.

First of all, an executing program requires that its instructions exist somewhere in memory. The segment of memory that holds the currently executing sequence of instructions is called the current code segment; it is specified by the CS register. This code segment provides all instructions, using the contents of the instruction pointer (IP) as an offset. Thus, the CS:IP register combination makes up the full 32-bit pointer for the next sequential program instruction, and the CS register is manipulated indirectly. Transitions from one code segment to another (e.g., a procedure call) are effected implicitly because of control-transfer instructions, interrupts, and trap operations.

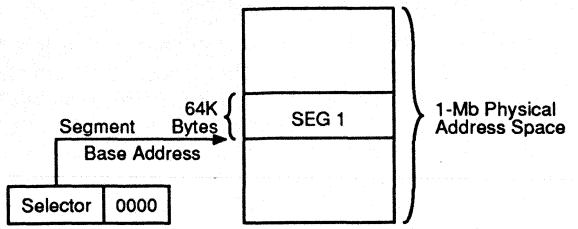
Stacks play an important role in the 80C286 architecture. For example, a number of implicit stack operations are involved in a subroutine call. As a result, an executing program generally requires a region of memory for its stack. The segment containing this region is called the current stack segment and is specified by means of the SS register. All stack operations are executed within this segment, usually in terms of address offsets contained in the stack pointer (SP) and stack frame base (BP) registers. The SS register, unlike the CS register, can be loaded explicitly for dynamic stack definition.

Most programs must also fetch and store data in memory beyond their code and stack requirements. The DS and ES registers provide for the specification of two data segments; the currently executing program can address each segment. This accessibility to two separate data areas supports differentiation and access requirements such as local procedure data and global process data. An operand within a data segment is addressed by specifying its offset either directly in an instruction or indirectly through index and/or base registers, which are described in the next subsection.

A program may require access to multiple data segments depending on the data structure and the way data is parceled into one or more segments. The DS and ES registers may be loaded under program control while executing a program in order to access additional segments. The appropriate data pointer must be loaded prior to accessing the data.

How segment selector values are interpreted depends on the operating mode of the processor. A segment selector is a physical address in real address mode (Figure 2-5). A segment selector selects a segment of the user's virtual address space in protected mode (Figure 2-6). An intervening level of logical-to-physical address translation transforms the logical address to a physical memory address. In general, considerations of selector formats and the details of memory mapping are not of concern for the application programmer.

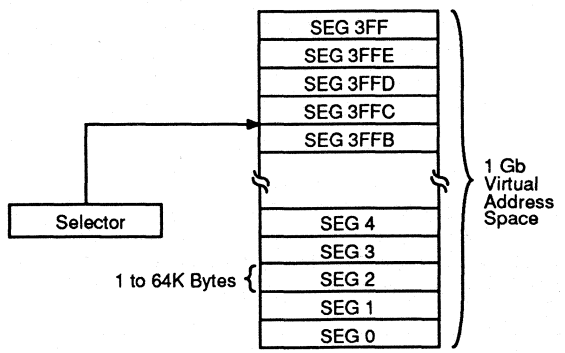
Figure 2-5 Real Address Mode Segment Selector Interpretation



Notes: 1. The selector identifies a segment in physical memory.
2. A selector specifies the segments base address, MODULO 16, within the 1 Megabyte address space.
3. The selector is the 16 most significant bits of a segments physical base address.
4. The values of selectors determines the amount they overlap in real memory.
5. Segments may overlap by increments of 16 bytes. Overlap ranges from complete (SEG 1 = SEG 1) to none (SEG 1 ± SEG 2 ± 64K).

14554A-004

Figure 2-6 Protected Mode Segment Selector Interpretation



Notes: 1. The selector uniquely identifies (names) one of 16K possible segments in the task's virtual address space.
2. The selector value does not specify the segment's location in physical memory.
3. The selector does not imply any overlap with other segments (this depends on the base address of the segment as specified via the memory management and protection information).

14554A-005

Index, Pointer, and Base Registers

Five of the general-purpose registers that are available for offset address calculations are shown in Figure 2-4. They are SP, BP, BX, SI, and DI. SP is a pointer register; BP and BX are base registers; and SI and DI are index registers.

As described above, segment registers define the set of four segments that may be currently addressed by a program. A pointer, base, or index register may contain an

offset value relative to the start of one of these segments. Thus, it points to the location of a particular operand within that segment. All base and index registers may work interchangeably as operands in most arithmetical operations to allow for efficient computations of effective address offsets.

The stack pointer (SP) and stack frame base (BP) registers facilitate stack operations. By specifying offsets into the current stack segment, both registers provide access to data on the stack. The SP register, referenced implicitly by PUSH and POP operations, subroutine calls, and interrupt operations, is the customary top-of-stack pointer. It addresses the uppermost datum on a push-down stack. The BP register provides still another offset into the stack segment. In conjunction with certain addressing modes, this stack relative base register is especially useful for accessing data structures, variables, and dynamically allocated work space within the stack.

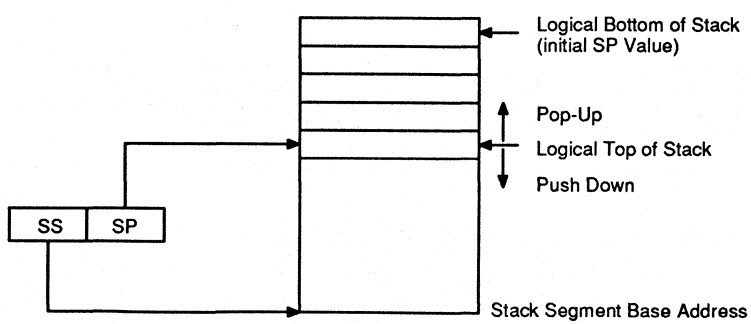
The stack segment register (SS) and the stack pointer register (SP) locate the stacks in the 80C286 that are implemented in memory. A system can have an unlimited number of stacks, and a stack can be up to the maximum length of a segment, 64K bytes.

Only the current stack, often referred to simply as the stack, is directly addressable. The current top of the stack (TOS) is contained in SP. In other words, the SP register contains the offset to the top of the push down stack from the stack segment's base address, but the stack's base address (contained in SS) is not the bottom of the stack (see Figure 2-7).

The stack frame base pointer (BP) is often used to access elements on the stack relative to a fixed point on the stack instead of the current TOS. It generally indicates the base address of the current stack frame established for the current procedure (Figure 2-8). The offset will be calculated automatically in the current stack segment if an index register is used relative to BP (e.g., base + index addressing mode using BP as the base).

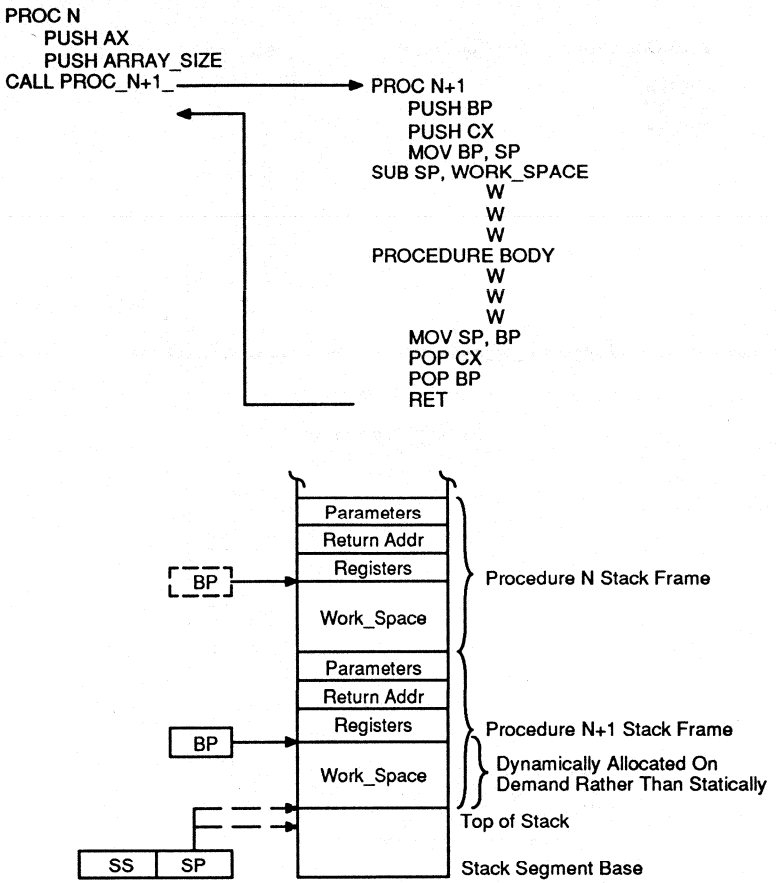
The BX register, which has the same function in addressing operands within data segments that BP does for stack segments, accesses data structures in data segments. They are known as base registers, because they may contain an offset to the base of a data structure. When discussing addressing modes, the similar usage of these two registers is particularly important.

Figure 2-7 **80C286 Stack**



14554A-006

Figure 2-8 **BP Usage as a Stack Frame Base Pointer**



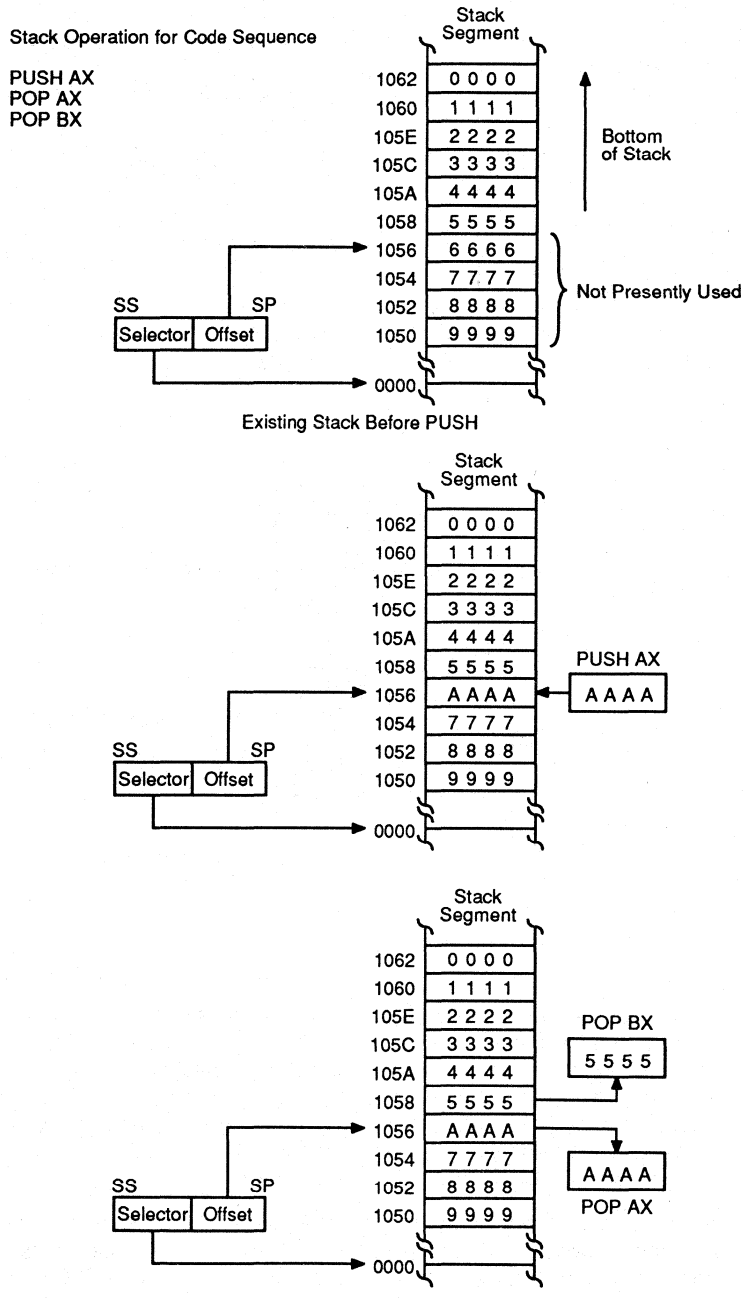
Note: BP is a constant pointer to stack based variables and work space. All references use BP and are independent of SP, which may vary during a routine execution.

14554A–008

80C286 stack entries are 16 bits wide, and instructions operate on the stack by adding and removing stack items one word at a time. An item is added (pushed) to the stack (see Figure 2-9) by decrementing SP by 2 and writing the item at the new TOS. An item is removed (popped) from the stack by copying it from TOS and then *incrementing* SP by 2. Therefore, the stack grows *down* in memory toward its base address. Stack operations never move or erase items on the stack. The top of the stack can change only by updating the stack pointer.

SI and DI registers also facilitate operations on data. By specifying an offset relative to the start of the currently addressable data segment, an index register can address an operand in the segment. If an index register is used with the BX base register (i.e., base + index addressing) to form an offset address, the data is assumed to reside in the current data segment. As a rule, data referenced through an index register, or BX, can be found in the current data segment. In other words, if an instruction invokes addressing for one of its operands using either BX, DI, SI, or BX with SI or DI, the

Figure 2-9 Stack Operation



14554A–009

contents of the register(s) (BX, DI, or SI) implicitly specify an offset in the current data segment. As mentioned earlier, data referenced by SP, BP or BP with SI or DI implicitly specify an operand in the current stack segment (see Table 2-1).

There are two exceptions to the rules discussed above. The first exception concerns the operation of certain 80C286 string instructions. For the most flexibility, these instructions assume that the DI register addresses destination strings in the extra segment (ES register) instead of in the data segment. Thus, movement of strings between different segments is possible and has led to the descriptive names of Source index and Destination index. However, in all instances other than string instructions, the SI and DI registers may be used interchangeably to reference either source or destination operands.

The second exception is a more general override capability that gives the programmer complete control over which segment is used for a specific operation. Segment-override prefixes let the index and base registers to address data in any of the four currently addressable segments.

Table 2-1 **Implied Segment Usage by Index, Pointer, and Base Registers**

Register	Implied Segment
SP	SS
BP	SS
BX	DS
SI	DS
DI	DS, ES for String Operations
BP + SI, DI	SS
BX + SI, DI	DS

Status and Control Registers

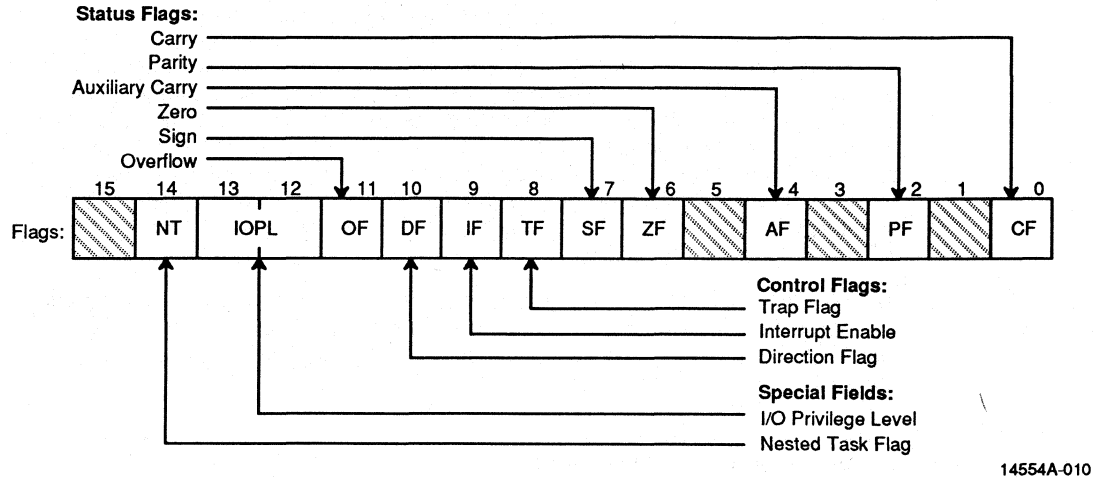
The instruction pointer and the FLAGS registers are two status and control registers that are of immediate concern to applications programmers.

The instruction pointer register (IP) contains the offset address, relative to the start of the current code segment, of the next sequential instruction to be performed. Together, the CS:IP registers make up a 32-bit program-counter. The instruction pointer is not directly visible to the programmer. Instead, it is controlled implicitly, by interrupts, traps, and control-transfer operations.

The FLAGS register contains eleven flag fields, most of them one-bit wide, as shown in Figure 2-10. The status flags, which record processor status information, comprise six of the eleven flags. The execution of arithmetic and logical instructions affects the status flags. The carry flag is modifiable with instructions to clear, set, or complement this flag bit.

The carry flag (CF) typically identifies a carry or borrow out of the most significant bit of an 8- or 16-bit operand after executing an arithmetic operation. This flag is also useful for bit manipulation operations that involve the shift and rotate instructions. The effect on the other status flags, when defined for a specific instruction, is usually as follows: the zero flag (ZF) indicates a zero result when set; the sign flag (SF) shows that the result was negative (SF = 1) or positive (SF = 0); when set, the overflow flag (OF) tells if an operation results in a carry into the high order bit of the result, but not a

Figure 2-10 Flags Register



carry out of the high-order bit, or vice versa; and the parity flag (PF) identifies whether the MODULO 2 sum of the low-order eight bits of the operation is even (PF = 0) or odd (PF = 1) parity. When performing binary coded decimal (BCD) arithmetic, the auxiliary carry flag (AF) signifies a carry out of or borrow into the least significant 4-bit digit.

Three control flags in the FLAGS register are used, under program control, to direct certain processor operations. If set, the interrupt-enable flag (IF) enables external interrupts; otherwise, they are disabled. The trap flag (TF), if set, puts the processor into a single-step mode for debugging purposes. As a result, the target program is automatically interrupted to a user supplied debug routine after the execution of each target program instruction. The direction flag (DF) controls the direction of string operations, 0 being forward or auto-increment the address register(s) (SI, DI or SI and DI), and 1 being backward or auto-decrement the address register(s) (SI, DI or SI and DI).

With special instructions (STI = set, CLI = clear), or by placing the flags on the stack, modifying the stack, and returning the flag image from the stack to the flag register, the interrupt enable flag may be set or reset. The ability to alter the IF bit, if operating in protected mode, is subject to protection checks in order to prevent non-privileged programs from effecting the interrupt state of the CPU. This is applicable to both instruction and stack options for modifying the IF bit.

The TF flag may be modified by:

1. Copying the flag register to the stack.
2. Setting the TF bit in the stack image.
3. Returning the modified stack image to the flag register.

The trap interrupt is executed upon completion of the next instruction, and entry to the single step routine saves the flag register on the stack with the TF bit set. The TF bit is then reset in the register. When the single step routine is finished, the TF bit is automatically set on return to the program being single stepped to interrupt the program again after the next instruction is completed. The protection mechanism in protected mode does not inhibit the use of TF.

Like the IF flag, the DF flag is manipulated by instructions (CLD = clear, STD = set) or flag register modification through the stack. Generally, routines using string instructions will save the flags on the stack, modify DF as needed through provided instructions, and restore DF to its original state by restoring the flag register from the stack before returning. The protection mechanism in protected mode does not inhibit access or control of the DF flag.

The Special Fields bits are of concern only in protected mode. Real address mode programs should treat these bits as DON'T CARE's and make no assumption regarding their status. Attempts to change the IOPL and NT fields may elicit protection checking in protected mode. In general, the application's programmer will be unable to and thus should not attempt to modify these bits.

ADDRESSING MODES

The information encoded in an 80C286 instruction consists of a specification of the operation to be performed, and the type of the operands to be manipulated, including their location. If an operand is in memory, the instruction must also choose, explicitly or implicitly, which currently addressable segment contains the operand. This section discusses the operand addressing mechanisms; Chapter 3 discusses 80C286 operators.

The five elements of a general instruction are listed below. The exact format of 80C286 instructions is found in Appendix B.

- The opcode, present in all instructions, is the only required element. It specifies the operation performed by the instruction.
- A register specifier.
- The addressing mode specifier, when present, specifies the addressing mode of an operand to reference data or perform indirect calls or jumps.
- The displacement, when present, computes the effective address of an operand in memory.
- The immediate operand, when present, directly identifies one operand of the instruction.

Of the four elements, only one is always present: the opcode. The presence of the other elements depends on the particular operation involved and on the location and type of the operands.

Operands

Generally speaking, an instruction is an operation executed on zero, one, or two operands, which are the data controlled by the instruction. An operand can be either in a register (AX, BX, CX, DX, SI, DI, SP, or BP for 16-bit operands; AH, AL, BH, BL, CH, CL, DH, or DL for 8-bit operands; the FLAG register for flag operations in the instruction itself (as an immediate operand)), in memory, or in an I/O port. Because immediate and register operands are available in the processor, they can be addressed quicker than operands in memory.

An 80C286 instruction can refer to zero, one, or two operands as follows:

- Zero-operand instructions, such as RET, NOP, and HLT. See Appendix B.
- One-operand instructions, such as INC or DEC. The location of the single operand can be specified *implicitly,* as in AAM (where the register AX holds the

operand), or *explicitly,* as in INC (where the operand may be in any register or memory location). Access to explicitly specified operands is possible through one of the addressing modes.

- Two operand instructions such as MOV, ADD, XOR, etc. These usually overwrite one of the two participating operands with the result. Therefore, a distinction can be made between the source operand (the one unaffected by the operation) and the destination operand (the one overwritten by the result). Like one-operand instructions, two-operand instructions may either explicitly or implicitly specify the location of operands. If an instruction contains two explicitly specified source and destination operands, only one can be in a register or memory location. The other must be contained in a register or be an immediate source operand. Special instances of two-operand instructions are the string instructions and stack manipulation. Some string instructions have both operands in memory and thus are explicitly specified. Transfer between memory operands and the memory based stack is allowed by push and pop stack operations.

The two-operand instructions of the 80C286 allow operations of the following sort:

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

Instructions may indicate the location of their operands with eight addressing modes.

Register and Immediate Modes

Two addressing modes reference operands located in registers and instructions:

- Register Operand Mode. The operand is contained in one of the 16-bit 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). There are special instructions for referencing the CS, DS, ES, SS, and Flag registers as operands.
- Immediate Operand Mode. The operand is part of the actual instruction (the immediate operand element).

Memory Addressing Modes

Six modes can access operands in memory. Memory operands are accessed by a pointer consisting of a segment selector and an offset, which specifies the operand's displacement in bytes from the beginning of the segment in which it is contained. Both the segment selector and the offset components are 16-bit values. Only certain instructions use a complete 32-bit address.

Most memory references do not need the instruction in order to specify a full 32-bit pointer address. Operands within one of the currently addressable segments, as determined by the four segment registers, can be referenced very efficiently by the 16-bit offset. This address form is called a short address. The segment choice (CS, DS, ES, or SS) is either implicit within the instruction itself or explicitly specified by a segment override prefix.

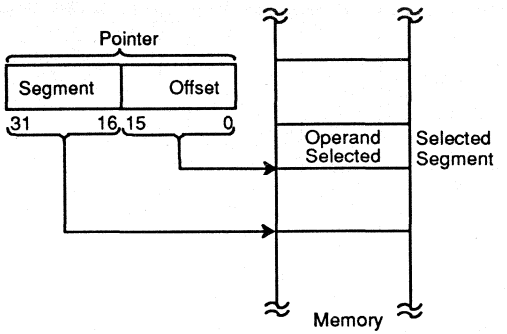
Figure 2-11 provides a diagram of the addressing process.

SEGMENT SELECTION

All instructions addressing operands in memory must specify the segment and the offset. For speed and compact instruction encoding, typically the high speed segment registers store segment selectors. Only the desired segment register and an offset must be specified by an instruction to address a memory operand.

Most instructions are not required to explicitly specify which segment register is used. Instead, the correct segment register is automatically selected according to the rules in Tables 2-1 and 2-2. These rules follow how programs are written as independent modules that require areas for code and data, a stack, and access to external data areas (see Figure 2-12).

Figure 2-11 **Two-Component Address**



09729B-004

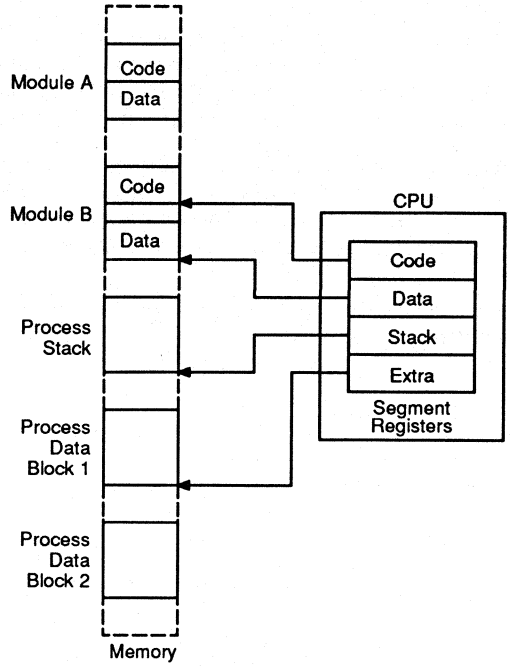
Table 2-2 **Segment Register Selection Rules**

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 which uses BP as a base register.
Local Data	Data (DS)	All data references except when relative to stack or string destination.
External (Global) Data	Extra (ES)	Alternate data segment and destination of string operation.

A close connection exists between the type of memory reference and the segment where that operand resides. Generally, a memory reference implies the current data segment (i.e., the implicit segment selector is in DS). If the BP register is involved in the address specification, the current stack segment is implied (i.e., SS contains the selector).

Special instruction prefix elements are defined in the 80C286 instruction set (see Appendix B). SEG, the segment-override prefix, is one of these elements. Segment-override prefixes provide for explicit segment selection. An implied segment selection that cannot be overridden exists only in two special cases—the use of DI to reference destination strings in the ES segment, and the use of SP to reference stack locations in the SS segment. Appendix B shows the format of segment override prefixes.

Figure 2-12 Use of Memory Segmentation



14554A-011

OFFSET COMPUTATION

The offset within the desired segment is calculated in accordance with the desired addressing mode by taking the sum of up to three components:

- The instruction's displacement element
- The base (contents of BX or BP—a base register)
- The index (contents of SI or DI—an index register)

Each of these offset components may be either a positive or negative value, and offsets are calculated MODULO 2^{16}.

Combinations of these three components generate the six memory addressing modes. The six modes access different types of data stored in memory:

Addressing Mode	Offset Calculation
Direct Address	Displacement Alone
Register Indirect	Base or Index Alone
Based	Base + Displacement
Indexed	Index + Displacement
Based Indexed	Base + Index
Based Indexed with Displacement	Base + Index + Displacement

In all six modes, the operand is within the selected segment at the specified offset. With the exception of direct address mode, all displacements are optionally 8- or

16-bit values, and 8-bit displacements are automatically sign-extended to 16 bits. The following section on memory addressing modes describes and demonstrates the six addressing modes.

MEMORY MODE

Two of the six modes are for simple scalar operands in memory:

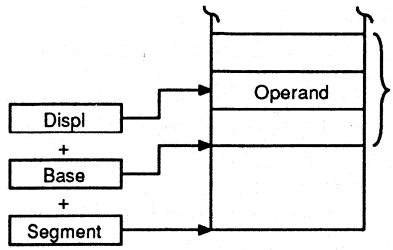
- Direct Address Mode. The operand's offset, a 16-bit quantity, is contained in the instruction as the displacement element.
- Register Indirect Mode. The offset of the operand is contained in SI, DI, or BX. (BP is excluded because if used as a stack frame base, it needs an index or displacement component to reference either parameters passed on the stack or temporary variables allocated on the stack. The instruction level bit encoding for the BP only address mode specifies Direct Address mode.)

The next four modes access complex data structures in memory (see Figure 2-13):

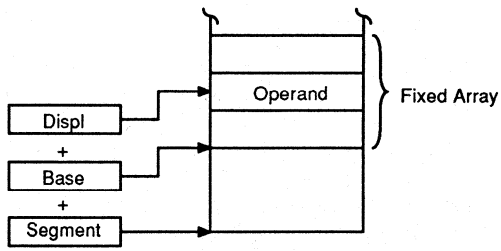
- Based Mode. The operand is located within the selected segment at an offset that is the sum of the displacement and the contents of a base register (BX or BP). Based mode can access the same field in different copies of a structure (also known as a record). The base register points to the structure's base (thus the term base register), and the displacement chooses a certain field. By changing the base register, corresponding fields within a collection of structures can be easily accessed. (See Figure 2-13, example 1.)
- Indexed Mode. The operand is located within the selected segment at an offset that is the sum of the displacement and the contents of an index register (SI or DI). This mode can access elements in a static array (an array with a starting location that is fixed at translation time). The displacement finds the beginning of the array, and the value of the index register chooses one element. Since all array elements are of equal length, simple arithmetic on the index register will pick an element. (See Figure 2-13, example 2.)
- Based Indexed Mode. The operand is located within the selected segment at an offset that is the sum of the base register's contents and an index register's contents. Based Indexed mode can access elements of a dynamic array (an array with a base address that can change during execution). The base register points to the array's base, and the index register value is used to select one element. (See Figure 2-13, example 3.)
- Based Indexed Mode with Displacement. The operand is located with the selected segment at an offset that is the sum of a base register's contents, an index register's contents, and the displacement. This mode can access elements of an array within a structure. The structure, for instance, could be an activation record (a region of the stack that holds the register contents, parameters, and variables associated with one instance of a procedure). One variable could be an array. The base register points to the beginning of the activation record, the displacement indicates the distance from the start of the record to the beginning of the array variable, and the index register chooses a specific element of the array. (See Figure 2-13, example 4.)

Figure 2-13 Complex Addressing Modes

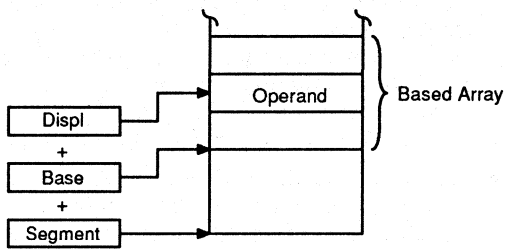
1. Based Mode
MOV AX, [BP + DATE_CODE]
ADD [BX + BALANCE], CX



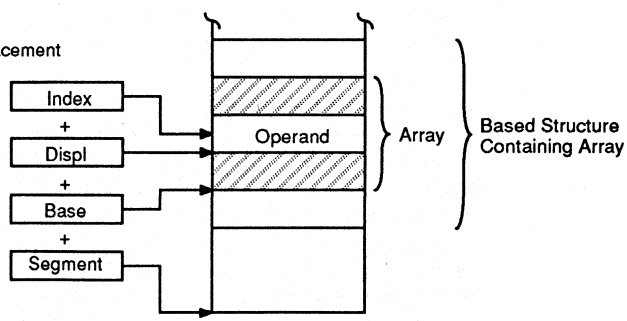
2. Indexed Mode
MOV ID, [SI], DX
SUB BX, DATA_TBL [SI]



3. Based Indexed
MOV DX, [BP], [DI]
AND [BX + S], 3FFH



4. Based Indexed Mode with Displacement
MOV CX, [BP], [SI + CNT]
SHR [BX + DI + MASK]



14554A–0012

Table 2-3 Memory Operand Addressing Modes

Addressing Mode	Offset Calculation
Direct	16-bit Displacement in the instruction
Register indirect	BX, SI, DI
Based	(BX or BP) + Displacement*
Indexed	(SI or DI) + Displacement*
Based Indexed	(BX or BP) + (SI or DI)
Based Indexed + Displacement	(BX or BP) + (SI or DI) + Displacement*

* The displacement can be a 0, 8, or 16-bit value.

Table 2-3 summarizes all memory operand addressing options.

INPUT/OUTPUT

Input/output on the 80C286 may be executed in either of two ways: by a separate I/O address space, using specific I/O instructions, or by memory-mapped I/O, using general-purpose operand manipulation instructions.

I/O Address Space

The 80C286 supplies a separate I/O address space that is distinct from physical memory to address the input/output ports used for external devices. The I/O address space consists of 2^{16} (64K) individually addressable 8-bit ports, but two consecutive 8-bit ports can be treated as a 16-bit port. As a result, the I/O address space can accommodate up to 64K 8-bit ports or up to 32K 16-bit ports. I/O port addresses 00F8H to 00FFH are reserved.

The 80C286 can move 8 or 16 bits to a device located in the I/O space at a time. In order for the 16 bits to be transferred in a single access,16-bit ports should be aligned at even-numbered addresses, like words in memory. An 8-bit port, however, may be located at an even or odd address. The internal registers in a peripheral controller device should be assigned the addresses shown below.

Port Register	Port Addresses	Example
16-bit	Even word addresses	OUT FE,AX
8-bit; device on lower half of 16-bit data bus	Even byte addresses	IN AL,FE
8-bit; device on upper half of 16-bit data bus	Odd byte addresses	OUT FF,AL

The I/O instructions IN and OUT are provided to transfer data between I/O ports and the AX (16-bit I/O) or AL (8-bit I/O) general registers. The block I/O instructions INS and OUTS, transfer blocks of data between I/O ports and memory space. An operating system in protected mode may prevent a program from performing these I/O instructions. Otherwise, the function of the I/O instructions and the structure of the I/O space are the same for both modes of operation.

```
INS es:byte ptr [di], DX

OUTS DX, byte ptr [si]
```

IN and OUT instructions address I/O either directly to one of up to 256 port addresses, or indirectly through the DX register to one of up to 64K port addresses. Block I/O uses the DX register to indicate the I/O address, and it uses either SI or DI to determine the source or destination memory address. SI or DI are either incremented or decremented for each transfer, as specified by the direction bit in the flag word, while DX remains constant to select the I/O device.

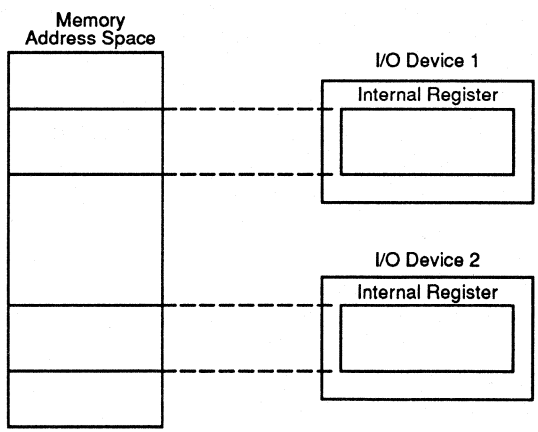
Memory-Mapped I/O

I/O devices, which may be placed in the 80C286 memory address space, are indistinguishable to the processor, so long as the devices respond like memory components.

The additional programming flexibility of memory-mapped I/O allows any instruction that references memory to access an I/O port located in the memory space. For example, the MOV instruction can move data between any register and a port. The AND, OR, and TEST instructions can manipulate bits in the internal registers of a device (see Figure 2-14). The full complement of addressing modes for choosing the desired I/O device is maintained by memory-mapped I/O performed by the full instruction set.

When executing in protected mode, memory-mapped I/O is subject to access protection and control like any other memory reference.

Figure 2-14 **Memory-Mapped I/O**



INTERRUPTS AND EXCEPTIONS

There are several mechanisms in the 80C286 architecture for interrupting program execution. Internal interrupts are the CPU's synchronous responses to particular events detected during instruction execution. External interrupts are asynchronous events typically set off by external devices that require attention. The 80C286 will support both maskable interrupts (controlled by the IF flag) and non-maskable interrupts. These interrupts cause the processor to service the requesting device by temporarily suspending its present program execution. The major difference between

these two kinds of interrupts lies in their origin: an internal interrupt can be reproduced by re-executing the program and data that caused it, but an external interrupt is usually independent of the currently executing task. Interrupts 31-0 are reserved. Servicing external interrupts will not generally concern application programmers.

In real address mode, two kinds of internal interrupts affect the application programmer. (Internal interrupts result from performing an instruction that causes the interrupt.) One type of interrupt, called an exception, only occurs if a certain fault condition exists. The other type of interrupt is generated each time the instruction is executed.

Divide error, INTO detected overflow, bounds check, segment overrun, invalid operation code, and processor extension error are exceptions (see Table 2-4). A divide error exception results from executing DIV or IDIV with a zero denominator; otherwise, the quotient will be too large for the destination operand. When the INTO instruction is executed and the OF flag is set (after an arithmetic operation that set the overflow (OF) flag), an overflow exception results. When the BOUND instruction is performed and the array index it checks falls outside the bounds of the array, a bounds check exception occurs. The segment overrun exception occurs when a word memory reference is attempted that extends beyond the end of a segment. An invalid operation code exception results from an attempt to execute an undefined instruction operation code. A processor extension error occurs when a processor extension detects an illegal operation.

An internal interrupt is generated whenever the instruction INT is performed. The interrupt handler routines, provided by the application program or as part of the system software (from the system programmers), determine the effects of this interrupt and all interrupts. Several other fault conditions are detected in protected mode, and result in internal interrupts.

HIERARCHY OF INSTRUCTION SETS

The 80C286 instruction set is divided into three separate descriptive subsets: the Basic Instruction Set, the Extended Instruction Set, and the System Control Instruction Set. The hierarchy of instruction sets defined by this division clarifies the relationships between the different processors in the 8086 family (see Figure 2-15).

The Basic Instruction Set is the common subset of instructions found on all 8086 family processors. Instructions for logical and arithmetic operations, data movement, input/output, string manipulation, and transfer of control are included.

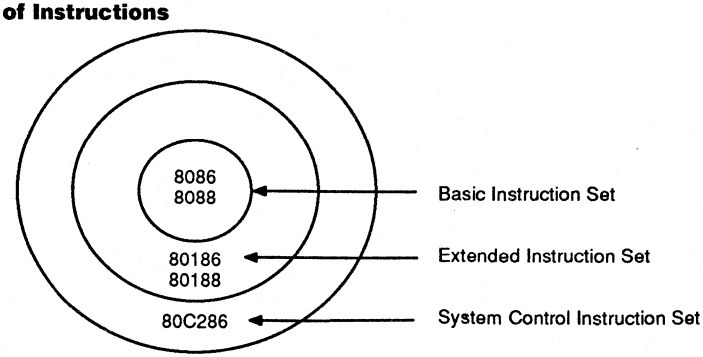
The Extended Instruction Set contains instructions found only on the 80186, 80188, and 80C286 processors. These instructions discuss block structured procedure entry and exit, parameter validation, and block I/O transfers.

Instructions unique to the 80C286 are found in the System Control Instruction Set. These instructions control the 80C286 memory management and protection mechanisms.

Table 2-4 **80C286 Interrupt Vector Assignments (Real Address Mode)**

Function	Interrupt	Related Instructions	Return Address Before Instruction
Divide error exception	0	DIV, IDIV	Yes
Single-step interrupt	1	All	
NMI interrupt	2	All	
Breakpoint interrupt	3	INT	
INTO detected overflow exception	4	INTO	No
BOUND range exceeded exception	5	BOUND	Yes
Invalid opcode exception	6	Any undefined opcode	Yes
Processor extension not available exception	7	ESC or WAIT	Yes
Interrupt table limit too small exception	8	INT vector is not within table limit	Yes
Processor extension segment overrun interrupt	8 9	ESC with memory operand extending beyond offset FFFF(H)	Yes
Reserved	10–12		
Segment overrun exception	13	Word memory reference with offset = FFFF(H) or an attempt to execute past the end of a segment	Yes
Reserved	14, 15		
Processor extension error	16	ESC or WAIT	
Reserved	17–31		
User defined	32–255		

Figure 2-15 **Hierarchy of Instructions**



CHAPTER 3

BASIC INSTRUCTION SET

The base architecture of the 80C286 maintains the complete instruction set of the 8086, 8088, 80188, and 80186 processors. The 80C286 instruction set contains new forms of some instructions, which reduce program size and improve the performance and ease of implementation of source code.

This chapter discusses instructions that programmers can use to write application software for the 80C286. The chapters that follow describe the operation of more complex I/O and system control instructions.

All instructions described in this chapter exist for both real address mode and protected virtual address mode operation. The instruction descriptions indicate any differences between the operation of an instruction in these two modes.

The operation of each application program-relative instruction and an example of using the instruction are presented in this chapter. Formal descriptions of all instructions are found in the Instruction Dictionary in Appendix B. Any opcode pattern that is not described in this dictionary is undefined and the result is an Opcode Violation trap (Interrupt 6).

DATA MOVEMENT INSTRUCTIONS

These instructions relay convenient methods for transferring bytes or words of data between memory and the registers of the base architecture.

General-Purpose Data Movement Instructions

MOVE (MOV)

MOV relocates a byte or a word from the source operand to the destination operand. The MOV instruction can transfer data to a register from memory, to memory from a register, between registers, immediate-to-register, or immediate-to-memory. Memory-to-memory or segment register-to-segment register moves are prohibited.

Example: MOV DS,AX. Moves the contents of register AX to replace the contents of register DS.

EXCHANGE (XCHG)

XCHG trades the contents of two operands and takes the place of three MOV instructions. A temporary memory location is not needed to save the contents of one operand while the other is being loaded.

Although the XCHG instruction can swap two byte operands or two word operands, it cannot trade a byte for a word or a word for a byte. Two register operands or a register operand with a memory operand may be the operands for the XCHG instruction. The XCHG instruction automatically activates the LOCK signal when used with a memory operand.

Example: XCHG BX,WORDOPRND. Exchanges the contents of register BX with the contents of the memory word designated by the label WORDOPRND after asserting bus lock.

Stack Manipulation Instructions

PUSH

PUSH decrements the stack pointer (SP) by two and then moves a word from the source operand to the top of stack indicated by SP (see Figure 3-1). PUSH, although often used to place parameters on the stack before calling a procedure, is also the basic means of storing temporary variables on the stack. This instruction operates on memory operands, immediate operands (new with the 80286), and register operands that include segment registers.

Example: PUSH WORDOPRND. Moves a 16-bit value from the memory word designated by the label WORDOPRND to the memory location that represents the current top of stack. Byte transfers are not allowed.

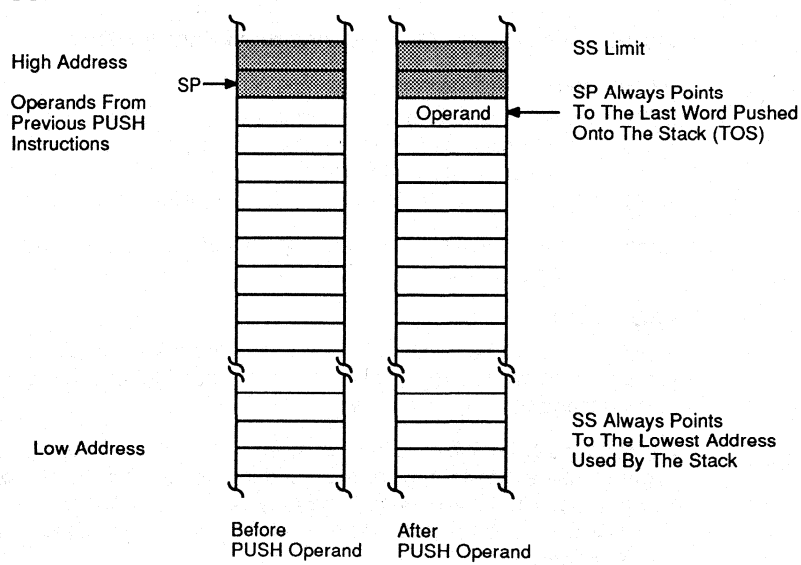
PUSH ALL REGISTERS (PUSHA)

PUSHA saves the contents of the stack's eight general registers (see Figure 3-2). The PUSHA instruction reduces the number of instructions required to retain the contents of the general registers for use in a procedure, and thus simplifies procedure calls. PUSHA works with the POPA instruction.

The general registers are pushed on the stack by the processor in the following order: AX, CX, DX, BX, the initial value of SP before AX was pushed, BP, SI, and DI.

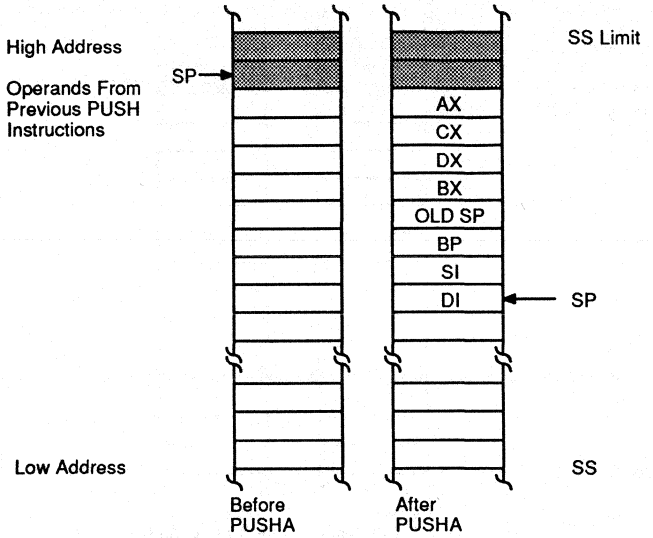
Example: PUSHA. Pushes the contents of the eight general registers onto the stack.

Figure 3-1 **PUSH**



PUSH decrements SP by 2 bytes and places the operand in the stack at the location to which SP points. 14554A–014

Figure 3-2 PUSHA



PUSHA copies the contents of the eight general registers to the stack in the above order. The instruction decrements SP by 16 bytes (8 words) to point to the last word pushed on the stack.

14554A–015

POP

POP moves the word at the current top of stack (indicated by SP) to the destination operand. It then increments SP by two to indicate the new top of stack (see Figure 3-3). The POP instruction transfers information from the stack to either a register or memory. POP, however, cannot place a value in register CS.

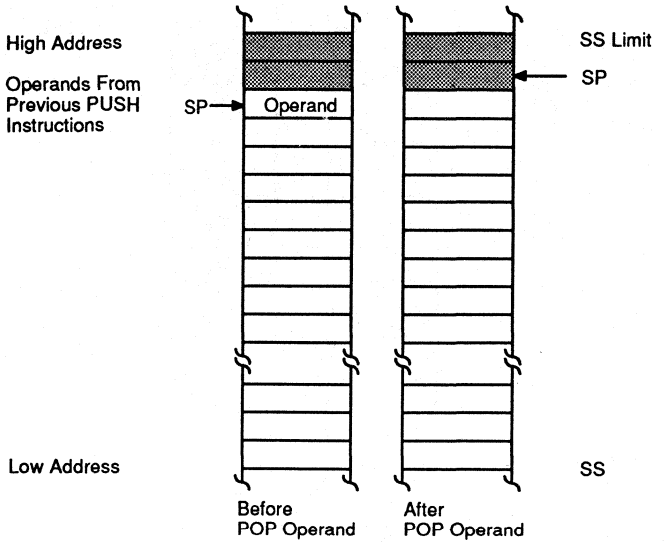
Example: POP BX. Moves the contents of the memory location at the top of the stack to replace the contents of register BX.

POP ALL REGISTERS (POPA)

POPA restores the registers saved on the stack by PUSHA, except that it disregards the value of SP (see Figure 3-4).

Example: POPA. Pops the saved contents of the general registers from the stack, and restores all registers (except SP) to their original state.

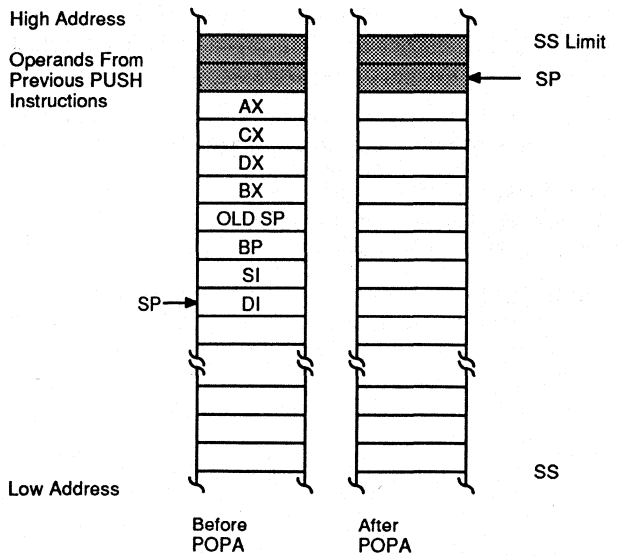
Figure 3-3 **POP**



POP copies the contents of the stack location before SP to the operand in the instruction.
POP then increments SP by 2 bytes (1 word).

14554A–015

Figure 3-4 **POPA**



POPA copies the contents of seven stack locations to the corresponding general registers.
POPA discards the stored value of SP.

14554A–016

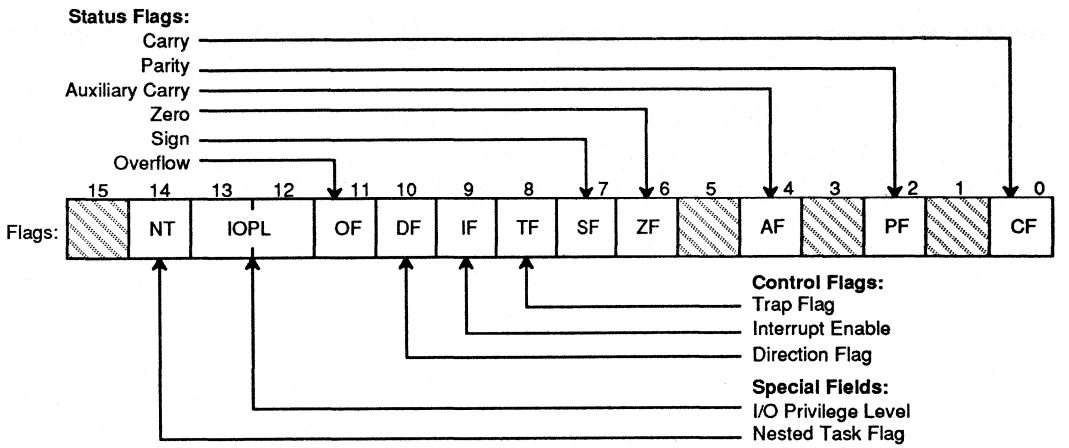
FLAG OPERATION WITH THE BASIC INSTRUCTION SET

Status Flags

Conditions that result from a previous instruction or instructions are reflected by the status flags of the FLAGS register. The arithmetic instructions utilize OF, SF, ZF, AF, PF, and CF.

The Scan String (SCAS), Compare String (CMPS), and LOOP instructions use ZF to indicate that their operations are complete. The base architecture of the 80C286 contains instructions to set, clear, and complement CF before executing an arithmetic instruction (see Figure 3-5 and Tables 3-1 and 3-2).

Figure 3-5 **Flag Word Contents**



14554A-017

Table 3-1 **Status Flags' Functions**

Bit Position	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	Set on carry from or borrow to the low order four bits of AL; cleared otherwise.
6	ZF	Zero Flag—Set if result is zero; cleared otherwise.
7	SF	Sign Flag—Set equal to high-order bit of result (0 if possible, 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.

Table 3-2 **Control Flags' Functions**

Bit Position	Name	Function
8	TF	Trap (Single Step) Flag—Once set, a single step interrupt occurs after the next instruction executes. TF is cleared by the single step interrupt.
9	IF	Interrupt-Enable Flag—When set, maskable interrupts will cause the CPU to transfer control to an interrupt vector-specified location.
10	DF	Direction Flag—Causes string instructions to auto decrement the appropriate index registers when set. Clearing DF causes auto decrement.

Control Flags

The control flags of the FLAGS register provide processor operations for string instructions, maskable interrupts, and debugging.

DIRECTION FLAG (DF)

Setting DF makes string instructions auto-decrement. That is, the direction flag processes strings from high addresses to low addresses, or from right-to-left. Clearing DF makes string instructions auto-increment, or process strings from left-to-right.

INTERRUPT FLAG (IF)

Setting IF lets the CPU recognize external (maskable) interrupt requests. By clearing IF, these interrupts are disabled. IF does not effect internally generated interrupts, nonmaskable external interrupts, or processor extension segment overrun interrupts.

TRAP FLAG (TF)

Setting TF places the processor in single-step mode for debugging. The CPU automatically generates an internal interrupt after each instruction in this mode. Thus, a program can be inspected as it executes each instruction.

ARITHMETIC INSTRUCTIONS

The 8086-family processors' arithmetic instructions facilitate the manipulation of numerical data with ease. Multiplication and division instructions simplify the handling of signed and unsigned binary integers as well as unpacked decimal integers.

An arithmetic operation may include two register operands, a general register source operand with a memory destination operand, a memory source operand with a register destination operand, or an immediate field with either a register or memory destination operand, but not two memory operands. Arithmetic instructions can be executed on either byte or word operands.

Addition Instructions

ADD INTEGERS (ADD)

ADD supersedes the destination operand with the sum of the source and destination operands. ADD operates on OF, SF, AF, PF, CF, and ZF.

Example: ADD BL, BYTEOPRND. Adds the contents of the memory byte labeled BYTEOPRND to the contents of BL, and replaces BL with the result.

ADD INTEGERS WITH CARRY (ADC)

ADC takes the sum of the operands, adds one if CF is set, and replaces the destination operand with the result. ADC is able to add numbers longer than 16 bits. ADC operates on OF, SF, AF, PF, CF, and ZF.

Example: ADC BX, CX. Replaces the contents of the destination operand BX with the sum of BX, CS, and 1 (if CF is set). ADC performs the same operation as the ADD instruction if CF is cleared.

INCREMENT (INC)

INC increases the destination operand by one. The processor regards the operand as an unsigned binary number. INC affects AF, OF, PF, SF, and ZF, but not CF. ADD should be used with an immediate value of 1 if an increment that updates carry (CF) is needed.

Example: INC BL. Adds 1 to the contents of BL.

Subtraction Instructions

SUBTRACT INTEGERS (SUB)

SUB subtracts the source operand from the destination operand and supersedes the destination operand with the result. Carry flag is set if a borrow is necessary. The operands can be signed or unsigned bytes or words. SUB operates on OF, SF, ZF, AF, PF, and CF.

Example: SUB WORDOPRND, AX. Replaces the contents of the destination operand WORDOPRND with the result generated by subtracting the contents of AX from the contents of the memory word labeled WORDOPRND.

SUBTRACT INTEGER WITH BORROW (SBB)

SBB subtracts the source operand from the destination operand, subtracts 1 if CF is set, and replaces the destination operand with the result. The operands can be signed or unsigned bytes or words. SBB is able to subtract numbers longer than 16 bits. This instruction operates on OF, SF, ZF, AF, PF, and CF. The carry flag is set if a borrow is needed.

Example: SBB BL, 32. Subtracts 32 from the contents of BL and then decrements the result by one if CF is set. SBB performs the same operation as SUB if CF is cleared.

DECREMENT (DEC)

DEC decreases the destination operand by one. DEC affects AF, OF, PF, SF, and ZF, but not CF. SUB should be used with an immediate value of 1 to execute a decrement that affects carry.

Example: DEC BX. Subtracts 1 from the contents of BX and replaces BX with the result.

Multiplication Instructions

UNSIGNED INTEGER MULTIPLY (MUL)

MUL executes an unsigned multiplication of the source operand and the accumulator. The processor multiplies the source, if it is a byte, by the contents of AL, and returns the double-length result to AH and AL.

The processor multiplies the source operand, if it is a word, by the contents of AX and returns the double-length result to DX and AX. CF and OF are set by MUL to indicate that the upper half of the result is not zero; otherwise, they are cleared. This instruction does not define SF, ZF, AF, and PF.

Example: MUL BX. Replaces the contents of DX and AX with the product of BX and AX. The low-order 16 bits of the result replace the contents of AX, and the high-order word replaces DX. If the unsigned result is greater than 16 bits, the processor sets CF and OF.

SIGNED INTEGER MULTIPLY (IMUL)

IMUL executes a signed multiplication operation. IMUL uses AX and DX like the MUL instruction, except when used in the immediate form.

The immediate form of IMUL provides for the specification of a destination register other than the combination of DX and AX. The result in this case cannot be greater than 16 bits without causing an overflow. Before performing the multiplication, the processor automatically extends an immediate operand that is a byte to 16 bits.

The immediate form of IMUL may also be used with unsigned operands because the low 16 bits of a signed or unsigned multiplication of two 16-bit values will always be identical.

IMUL clears CF and OF to show that the upper half of the result is the sign of the lower half. This instruction does not define SF, ZF, AF, and PF.

Example: IMUL BL. Replaces the contents of AX with the product of BL and AL. If the result is more than 8 bits long, the processor sets CF and OF.

Example: IMUL BX, SI, 5. Replaces the contents of BX with the product of the contents of SI and the immediate value of 5. If the signed result is longer than 16 bits, the processor sets CF and OF.

Division Instructions

UNSIGNED INTEGER DIVIDE (DIV)

DIV executes an unsigned division of the accumulator by the source operand. If the source operand is a byte, it is partitioned into the double-length dividend thought to be in registers AL and AH (AH being the most significant byte, and AL the least significant byte). The single-length quotient replaces AL, and the single-length remainder replaces AH.

If the source operand is a word, it is divided into the double-length dividend that is in registers AX and DX. The single-length quotient replaces AX, and the single-length remainder replaces DX. Non-integral quotients are truncated to integers toward 0, and the remainder is always less than the quotient.

The largest quotient for unsigned byte division is 255. The largest quotient for unsigned word division is 65,535. DIV does not define OF, SF, ZF, AF, PF, and CF. Interrupt (INT 0) results if the divisor is zero or if the quotient is too great for AL or AX.

Example: DIV BX. Replaces the contents of AX with the unsigned quotient resulting from the doubleword value in DX and AX divided by BX. The contents of DX are replaced by the unsigned MODULO.

Example: DIV BL. Replaces the contents of AL with the unsigned quotient resulting from the word value in AX divided by BL. The contents of AH are replaced by the unsigned MODULO.

SIGNED INTEGER DIVIDE (IDIV)

IDIV executes a signed division of the accumulator by the source operand. IDIV uses the identical registers of the DIV instruction.

The maximum positive quotient for signed byte division is +127 and the minimum negative quotient is –128. The maximum positive quotient for signed word division is +32,767 and the minimum negative quotient is –32,768. Non-integral results are truncated towards 0, and the remainder always has the same sign as the dividend and is less than the divisor in magnitude. IDIV does not define OF, SF, ZF, AF, PF, and CF. An interrupt (INT 0) occurs if the divisor is 0 or if the quotient is too great for AL or AX.

Example: IDIV WORDOPRND. Replaces the contents of AX with the signed quotient resulting from the double-word value in DX and AX divided by the value of the memory word labeled WORDOPRND. The contents of DX are replaced by the signed MODULO.

LOGICAL INSTRUCTIONS

The group of logical instructions contains the Boolean operation instructions, rotate and shift instructions, and type conversion and no-operation instructions.

Boolean Operation Instructions

Except for the NOT and NEG instructions, the Boolean operation instructions can use two register operands, a general purpose register operand with a memory operand, an immediate operand with a general purpose register operand, or a memory operand. The NOT and NEG instructions use a single operand in a register or memory.

AND

AND executes the logical AND of the operands (byte or word) and returns the result to the destination operand. AND clears OF and DF, updates SF, ZF, and PF, and does not define AF.

Example: AND WORDOPRND, BX. Replaces the contents of WORDOPRND with the logical AND of the memory word labeled WORDOPRND and BX.

NOT

NOT inverts the bits in the specified operand to create a one's complement of the operand. NOT does not affect the flags.

Example: NOT BYTEOPRND. Replaces the contents of BYTEOPRND with its one's complement.

OR

OR executes the logical Inclusive OR of the two operands and returns the result to the destination operand. OR clears OF and DF, updates SF, ZF, and PF, and does not define AF.

Example: OR AL,5. Replaces the contents of AL with the logical INCLUSIVE OR of those contents and the immediate value 5.

EXCLUSIVE OR (XOR)

XOR executes the logical EXCLUSIVE OR of the two operands and returns the result to the destination operand. XOR clears OF and DF, updates SF, ZF, and PF, and does not define AF.

Example: XOR DX, WORDOPRND. Replaces the contents of DX with the logical EXCLUSIVE OR of those contents and the contents of the memory word labeled WORDOPRND.

NEGATE (NEG)

NEG creates a two's complement of a signed byte or word operand. NEG reverses the sign of the operand from positive to negative or from negative to positive. NEG updates OF, SF, ZF, AF, PF, and CF.

Example: NEG AX. Replaces the contents of AX with the two's complement of those contents.

Shift and Rotate Instructions

The shift and rotate instructions relocate the bits within the specified operand. The shift instructions are a convenient way to divide or multiply by binary power, while the rotate instructions are helpful for bit testing.

SHIFT INSTRUCTIONS

The bits in bytes and words may be repositioned arithmetically or logically. Up to 31 shifts may be executed, depending on the value of a specified count.

A shift instruction can specify the count in one of three ways:

1. Implicitly specify the count as a single shift,
2. Specify the count as an immediate value,
3. Specify the count as the value contained in CL.

This last form lets the shift count be a variable that the program provides during execution, and only the low order 5 bits of CL are used.

Shift instructions also affect the flags. AF is never defined following a shift operation. PF, SF, and ZF are updated normally as they are in the logical instructions.

CF always includes 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. Nevertheless, the content of OF is never defined following a multi-bit shift.

Shift Arithmetic Left (SAL)

SAL moves the destination byte or word operand left either by one or by the number of bits specified in the count operand (an immediate value or the value in CL). The processor moves zeros in from the right side of the operand as bits leave from the left side (see Figure 3-6).

Example: SAL BL,2. Moves the contents of BL left by 2 bits and replaces the two low-order bits with zeros.

Example: SAL BL,1. Moves the contents of BL left by 1 bit and replaces the low-order bit with a zero. This form of the instruction takes 2 clock cycles rather than the 6 clock cycles (5 + 1 cycle for each bit shifted) required by the previous example, because the processor does not have to decode the immediate count operand to obtain the shift count.

Shift Logical Left (SHL)

SHL is the same instruction physically as SAL (see SAL above).

Shift Logical Right (SHR)

SHR moves the destination byte or word operand right either by one or by the number of bits specified in the count operand (an immediate value or the value in CL). The processor moves zeros in from the left side of the operand as bits leave from the right side (see Figure 3-7).

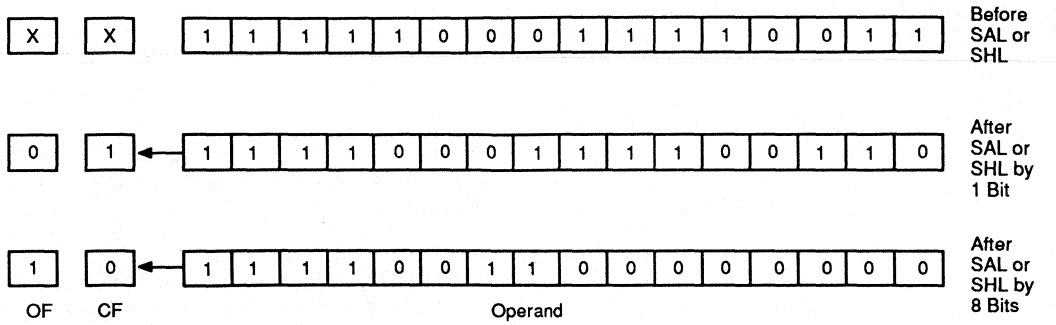
Example: SHR BYTEOPRND, CL. Moves the contents of the memory byte labeled BYTEOPRND right by the number of bits specified in CL, and puts the same number of zeros on the left side of BYTEOPRND.

Shift Arithmetic Right (SAR)

SAR moves the destination byte or word operand to the right either by one or by the number of bits specified in the count operand (an immediate value or the value in CL). The processor maintains the sign of the operand by moving in zeros on the left side if the value is positive or by moving in ones if the value is negative (see Figure 3-8).

Example: SAR WORDPRND,1. Moves the contents of the memory byte labeled WORDPRND to the right by one and replaces the high-order sign bit with a value equal to the original sign of WORDPRND.

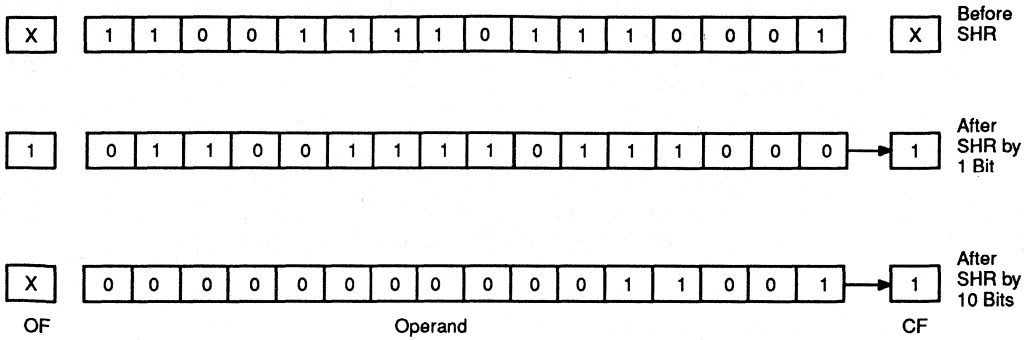
Figure 3-6 **SAL and SHL**



Both SAL and SHL shift 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. SAL and SHL shift in zeros to fill the vacated bit locations. These instructions operate on byte operands as well as word operands.

14554A-018

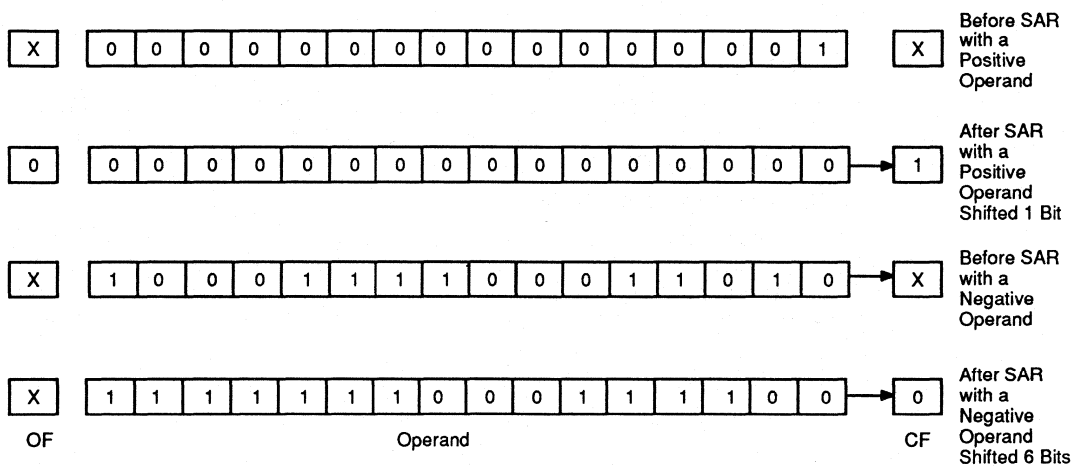
Figure 3-7 SHR



SHR shifts the bits in the register or memory operand to the right by the specified number of bit positions. CF receives the last bit shifted out of the left of the operand. SHR shifts in zeros to fill the vacated bit locations. These instructions operate on byte operands as well as word operands.

14554A-019

Figure 3-8 SAR



SAR preserves the sign of the register or memory operand as it shifts the operand to the right the specified number of bit positions. CF receives the last bit shifted out of the right of the operand. This instruction also operates on byte operands.

14554A-020

ROTATE INSTRUCTIONS

Bits in bytes and words may be moved without being lost through rotate instructions. Instead of bits being moved out of an operand, as in a shift, they are rotated, or circled back into the other end of the operand.

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

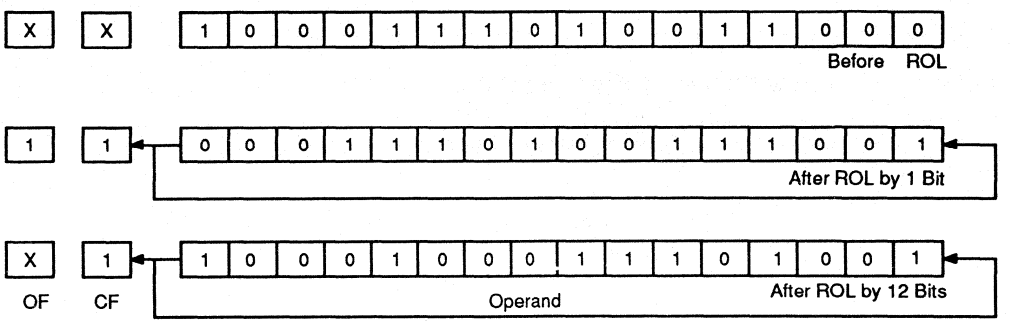
In single-bit rotates, OF is set if the operation alters the high-order (sign) bit of the destination operand. OF is cleared if the sign bit maintains its original value. The value of OF is never defined on multibit rotates.

Rotate Left (ROL)

ROL moves the byte or word destination operand left either by one or by the number of bits specified in the count operand (an immediate value or the value in CL). For each rotation specified, the high-order bit that leaves from the left of the operand returns at the right and becomes the new low-order bit of the operand (see Figure 3-9).

Example: ROL AL, 8. Moves the contents of AL left by 8 bits. In this rotate instruction, AL returns to its original state but the low-order bit is isolated in CF for testing by a JC or JNC instruction.

Figure 3-9 ROL



ROL shifts the bits in the memory or register operand to the left by the specified number of bit positions. It copies the bit shifted out of the left of the operand into the right of the operand. The last bit shifted into the least significant bit of the operand also appears in CF. This instruction also operates on byte operands.

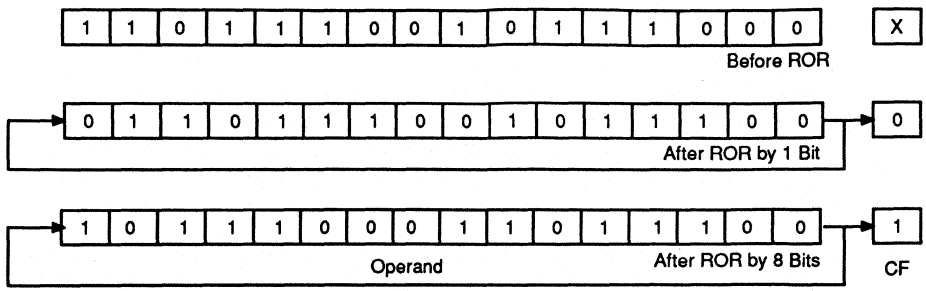
14554A-021

Rotate Right (ROR)

ROR moves the byte or word destination operand right either by one or by the number of bits specified in the count operand (an immediate value or the value in CL). For each rotation specified, the low-order bit that leaves from the right of the operand returns at the left and becomes the new high-order bit of the operand (see Figure 3-10).

Example: ROR WORDOPRND, CL. Moves the contents of the memory word labeled WORDOPRND by the number of bits specified by the value in CL. The value of the last bit rotated from the right to the left side of the operand is reflected by CF.

Figure 3-10 ROR



ROR shifts the bits in the memory or register operand to the right by the specified number of bit positions. It copies each bit shifted out of the right of the operand into the left of the operand. The last bit shifted into the most significant bit of the operand also appears in CF. This instruction also operates on byte operands.

14554A-022

Rotate Through Carry Left (RLC)

RCL moves bits in the byte or word destination operand to the left either by one or by the number of bits specified in the count operand (an immediate value or the value in CL).

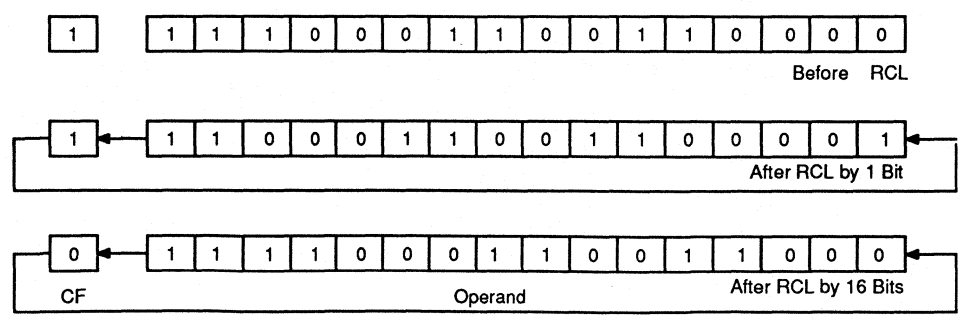
Unlike ROL, this instruction treats CF as a high-order 1-bit extension of the destination operand. Each high-order bit that leaves from the left side of the operand shifts to CF before it returns to the operand as the low-order bit on the next rotation cycle (see Figure 3-11).

Example: RCL BX,1. Moves the contents of BX to the left by one bit. The high-order bit of the operand shifts to CF, the remaining 15 bits shift left one position, and the original value of CF becomes the new low-order bit.

Rotate Through Carry Right (RCR)

RCR rotates bits in the byte or word destination operand to the right either by one or by the number of bits specified in the count operand (an immediate value or the value in CL).

Figure 3-11 RCL



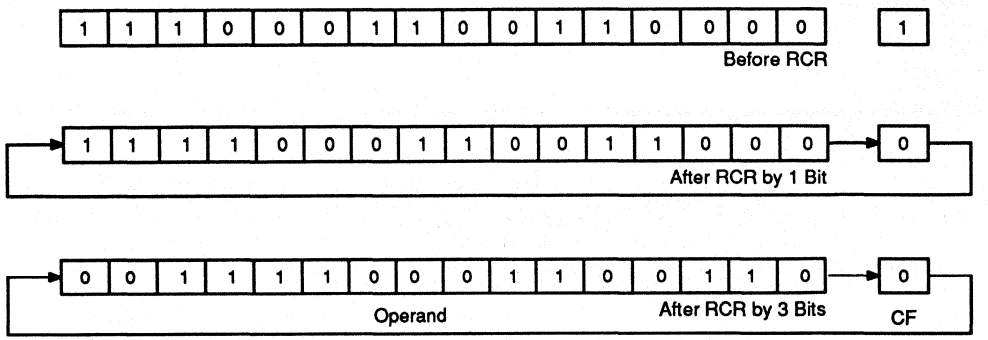
RCL rotates the bits in the memory or register operand to the left in the same way as ROL except that RCL treats CF as a 1-bit extension of the operand. Note that a 16-bit RCL produces the same result as a 1-bit RCR (though it takes much longer to execute). This instruction also operates on byte operands.

14554A-023

Unlike ROR, this instruction treats CF as a low-order 1-bit extension of the destination operand. Each low-order bit that leaves from the right side of the operand shifts to CF before returning to the operand as the high-order bit on the next rotation cycle (see Figure 3-12).

Example: RCR BYTEOPRND,3. Moves the contents of the memory byte labeled BYTEOPRND to the right by 3 bits. After this instruction is performed, CF becomes the original value of bit number 5 of BYTEOPRND, and the original value of CF becomes bit 2.

Figure 3-12 RCR



RCR rotates the bits in the memory or register operand to the right in the same way as ROR except that RCR treats CF as a 1-bit extension of the operand. This instruction also operates on byte operands.

14554A-024

Type Conversion and No-Operation Instructions

The type conversion instructions are used to prepare operands for division. The NOP instruction, a 1-byte filler instruction, does not affect registers or flags.

CONVERT WORD TO DOUBLE-WORD (CWD)

CWD extends the sign of the word in register AX throughout register DX. No flags are affected by CWD, which is able to produce a double-length (double-word) dividend from a word before a word division.

CONVERT BYTE TO WORD (CBW)

CBW extends the sign of the byte in register AL throughout AX. No flags are affected by CBW.

Example: CWD. Extends the sign of the 16-bit value in AX to a 32-bit value in DX and AX with the high-order 16 bits occupying DX.

NO OPERATION (NOP)

NOP occupies a byte of storage but affects only the instruction pointer, IP. The amount of time required by a NOP instruction for execution varies in proportion to the CPU clocking rate. Using NOP instructions in the construction of timing loops is not recommended because of this variation. The operation of such a program will not be independent of the system hardware configuration.

Example: NOP. The processor executes no operation for 2 clock cycles.

TEST AND COMPARE INSTRUCTIONS

The test and compare instructions do not change their operands. Instead, they execute operations that only set the appropriate flags to indicate the relationship between the two operands.

Test (TEST)

TEST executes the logical AND of the two operands, clears OF and DF, updates SF, ZF, and PF, and does not define AF. Unlike AND, TEST does not change the destination operand.

Example: TEST BL, 32. Executes a logical AND and sets SF, ZF, and PF according to the results of this operation. The contents of BL stay the same.

Compare (CMP)

CMPsubtracts the source operand from the destination operand. CMP updates OF, SF, ZF, AF, PF, and CF, but it does not change the source and destination operands. A subsequent signed or unsigned conditional transfer instruction is able to test the result using the appropriate flag result.

CMP is used to compare two register operands, a register operand and a memory operand, a register operand and an immediate operand, or an immediate operand and a memory operand. Although the operands may be words or bytes, CMP is unable to compare a byte with a word.

Example: CMP BX,32. Subtracts the immediate operand, 32, from the contents of BX and sets OF, SF, ZF, AF, PF, and CF according to the results of this operation. The contents of BX stay the same.

CONTROL TRANSFER INSTRUCTIONS

Both conditional and unconditional program transfer instructions are provided by the 80C286 to direct the flow of execution. While conditional program transfers are dependent on the results of operations that affect the flag register, unconditional program transfers are always executed.

Unconditional Transfer Instructions

JMP, CALL, RET, INT and IRET instructions shift control from one code segment location to another. These locations may be in either the same or different code segments.

JUMP (JMP)

JMP unconditionally shifts control to the target location. JMP, a one-way transfer of execution, does not save a return address on the stack.

The JMP instruction always transfers control from the current location to a new one. Its implementation depends on the following factors:

- Is the address indicated directly within the instruction or indirectly through a register or memory?
- Is the target location inside or outside the current code segment chosen in CS?

The destination address is included as part of a direct JMP instruction. The destination address is obtained indirectly through a register or a pointer variable in an indirect JMP instruction.

Control transfers through a gate or to a task state segment are available only in the 80C286's protected mode operation. The formats of the instructions that transfer control through a call gate, a task gate, or to a task state segment are identical. The label included in the instruction chooses one of these three paths to a new code segment.

Direct JMP Within the Current Code Segment

A direct JMP that shifts control to a target location within the current code segment uses a relative displacement value in the instruction, either a 16-bit value or an 8-bit value sign extended to 16 bits. To form an effective address, the processor adds this relative displacement tc the address in IP. When the additions are executed, IP refers to the next instruction.

Example: JMP NEAR_NEWCODE. Shifts control to the target location labeled NEAR_NEWCODE. This location is within the code segment currently selected in CS.

Indirect JMP Within the Current Code Segment

Indirect JMP instructions that shift control to a location within the current code segment indicate an absolute address in one of the following ways. First, the program can JMP to a location indicated by a 16-bit register (any of AX, DX, CX, BX, BP, SI, or DI). The processor transfers this 16-bit value into IP and then resumes execution.

Example: JMP SI. Shifts control to the target address, which is formed by adding the 16-bit value in SI to the base address in CS.

The processor can also get the destination address within a current segment from a memory word operand indicated in the instruction.

Example: JMP PTR_X. Shifts control to the target address, which is formed by adding the 16-bit value in the memory word labeled PTR X to the base address in CS.

A register can adjust the address of the memory word pointer to choose a destination address.

Example: JMP CASE_TABLE [BX]. CASE_TABLE is the first word in a group of word pointers. The value of BX decides which pointer the program chooses from the group. The JMP instruction then shifts control to the location that the chosen pointer indicates.

Direct JMP Outside of the Current Code Segment

Direct JMP instructions specifying a target location beyond the current code segment contain a full 32-bit pointer. This pointer includes a selector for the new code segment and an offset within the new segment.

Example: JMP FAR_NEWCODE_FOO. This instruction places the selector into CS and the offset into IP. Execution of the program resumes at this location in the new code segment.

Indirect JMP Outside of the Current Code Segment

Indirect JMP instructions specifying a target location beyond the current code segment contain a double-word variable to indicate the pointer.

Example: JMP NEWCODE. NEWCODE is the first word of two consecutive words in memory that represent the new pointer. NEWCODE includes the new offset for IP, and the word following NEWCODE includes the selector for CS. Execution of the program resumes at this location in the new code segment. (See Chapters 6 and 7 for a discussion of how protected mode programs treat this differently.)

Direct JMP Outside of the Current Code Segment to a Call Gate

If the selector within the instruction refers to a call gate, then the processor overlooks the offset in the instruction and uses the pointer of the routine that is entered from the call gate.

JMP beyond current code segment may only go to the same level.

Example: JMP CALL_GATE_FOO. The selector within the instruction refers to the call gate CALL_GATE_FOO, and the call gate actually supplies the new contents of CS and IP to indicate the address of the next instructions.

Indirect JMP Outside the Current Code Segment to a Call Gate

If the selector specified by the instruction refers to a call gate, the processor overlooks the offset in the double-word and uses the address of the routine that is entered from the call gate. To indirectly specify a task gate or a task state segment, the JMP instruction uses the identical format.

Example: JMP CASE_TABLE [BX]. The instruction refers to the double-word in the group of pointers called CASE_TABLE. The specific double-word selected is dependent on the value in BX when the instruction performs. The selector part of this double-word chooses a call gate, and the processor uses the address of the routine that is entered from the call gate.

CALL INSTRUCTION

CALL (Call Procedure) invokes an out-of-line procedure, saving on the stack the instruction's address following the CALL for later use by a RET (Return) instruction. An intrasegment CALL sets the current value of IP on the stack, while an intersegment CALL sets both the value of IP and CS on the stack. The RET instruction in the called procedure uses this address to shift control back to the calling program.

A long CALL instruction that activates a task-switch stores the outgoing task's task state segment selector in the incoming task state segment's link field and places the nested task flag in the new task. In this case, the IRET instruction replaces the RET instruction to return control to the nested task.

Examples:

```
CALL NEAR_NEWCODE
CALL SI
CALL PTR_X
CALL CASE_TABLE [BP]
CALL FAR_NEWCODE_FOO
CALL NEWCODE
CALL CALL_GATE_FOO
CALL CASE_TABLE [BX]
```

See the earlier treatment of JMP for more information on the operations of these instructions.

RETURN INSTRUCTIONS

Return from Procedure (RET)

RET ends a procedure and shifts control through a back-link on the stack to the program that originally activated the procedure.

An intrasegment RET restores the value of IP that was saved on the stack by the prior intrasegment CALL instruction. An intersegment RET restores the values of both CS and IP, which were saved on the stack by the prior intersegment CALL instruction.

RET instructions can specify a constant to the stack pointer. This constant indicates the new top of stack to effectively remove any arguments that the calling program pushed on the stack before the CALL instruction was executed.

Example: RET. RET restores the value of IP pushed by the CALL instruction if the prior CALL instruction did not shift control to a new code segment. If the prior CALL instruction shifted control to a new segment, RET restores the values of both IP and CS, which were pushed on the stack by the CALL instruction.

Example: RET n. This form of the RET instruction executes the same as the above example except that it adds n, which must be an even value, to the value of SP to remove n bytes of parameter information previously pushed by the calling program.

Return from Interrupt or Nested Task (IRET)

IRET restores control to an interrupted routine or reverses the action of a CALL or INT instruction that caused a task switch. Chapter 8 further discusses task switching.

Example: IRET. Returns from an interrupt with or without a task switch according to the value of the NT bit.

Conditional Transfer Instructions

The conditional transfer instructions are jumps that may or may not shift control, depending on the condition of the CPU flags when the instruction performs. Instruction encoding is optimum when the target for the conditional jumps is in the current code segment and within –128 to +127 bytes of the first byte of the following instruction. Optionally, the opposite sense of the conditional jump can go around an unconditional jump to the destination.

CONDITIONAL JUMP INSTRUCTIONS

Table 3-3 illustrates the conditional transfer mnemonics and their interpretations. The conditional jumps that are shown as pairs are actually the same instruction. For greater clarity within a program listing, the assembler provides alternate mnemonics.

LOOP INSTRUCTIONS

The loop instructions are conditional jumps that use a value set in CX to denote the number of repetitions of a software loop. All such instructions automatically decrement CX and stop the loop when CX = 0. Four of the five loop instructions specify a condition of ZF that ends the loop before CX decrements to zero.

Loop While CX not Zero (LOOP)

LOOP is a conditional transfer that auto-decrements the CX register prior to testing CX for the branch condition. If CX is non-zero, the program branches to the target label named in the instruction. The LOOP instruction repeats a code section until CX

Table 3-3 **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) = 0	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)

decrements to zero. If LOOP finds CX = 0, control shifts to the instruction immediately after the LOOP instruction. The LOOP executes 65,536 times if the value of CX is initially zero.

Example: LOOP START_LOOP. The program decrements CX and then tests it. If the value of CX is non-zero, then the program executes the instruction labeled START_LOOP. If the value in CX is zero, then the program executes the instruction after the LOOP instruction.

Loop While Equal (LOOPE) and Loop While Zero (LOOPZ)

LOOPE and LOOPZ are physically identical. These instructions auto-decrement the CX register prior to testing CX and ZF for the branch conditions. If CX is non-zero and ZF = 1, the program proceeds to the target label named in the instruction. If CX = 0 or ZF = 0, control shifts to the instruction immediately after the LOOPE or LOOPZ instruction.

Example: LOOPE START_LOOP (or LOOPZ START_LOOP). The program decrements CX and then tests CX and ZF. If the value in CX is non-zero and the value of ZF is 1, then the program executes the instruction labeled START_LOOP. If CX = 0 or ZF = 0, the program executes the instruction after the LOOPE (or LOOPZ) instruction.

Loop While Not Equal (LOOPNE) and Loop While Not Zero (LOOPNZ)

LOOPNE and LOOPNZ are physically identical. These instructions auto-decrement the CX register prior to testing CX and ZF for the branch conditions. If CX is non-zero and ZF = 0, the program proceeds to the target label named in the instruction. If

CX = 0 or ZF = 1, control shifts to the instruction immediately after the LOOPNE or LOOPNZ instruction.

Example: LOOPNE START_LOOP (or LOOPNZ START_LOOP). The program decrements CX and then tests CX and ZF. If the value of CX is non-zero and the value of ZF is 0, then the program executes the instruction labeled START_LOOP. If CX = 0 or ZF = 1, the program executes the instruction after the LOOPNE (or LOOPNZ) instruction.

Jump if CX is Zero (JCXZ)

JCXZ proceeds to the label named in the instruction if it finds a value of zero in CX. Occasionally, it is desirable to have a loop that does not execute if the count variable in CX is initialized to zero. Because the LOOP instructions (and repeat prefixes) decrement CX before they test it, a loop will perform 65,536 times if the initial value of CX is zero. A programmer may easily overcome this problem with JCXZ, which allows the program to proceed around the code within the loop if CX is zero when JCXZ performs.

Example: JCXZ TARGETLABEL. Causes the program to proceed to the instruction named TARGETLABEL if CX = 0 when the instruction performs.

Software-Generated Interrupts

The INT *n* and INTO instructions let the programmer indicate a control transfer to an interrupt service routine from within a program. Interrupts 0–31 are reserved.

INTERRUPT (INT *n*)

INT *n* invokes the interrupt service routine that corresponds to the number coded within the instruction. Interrupt type 3 is saved for internal software-generated interrupts. However, the INT instruction may select any interrupt type to allow for multiple types of internal interrupts or to test service routine operation. The interrupt service routine ends with an IRET instruction that restores control to the instruction following INT.

Example: INT 3. Shifts control to the interrupt service routine indicated by a type 3 interrupt.

Example: INT 0. Shifts control to the interrupt service routine indicated by a type 0 interrupt, which is saved for a divide error.

INTERRUPT ON OVERFLOW (INTO)

INTO activates a type 4 interrupt if OF is set when the INTO instruction performs. The type 4 interrupt is saved for this purpose.

Example: INTO. If the result of a prior operation has set OF and no intervening operation has reset OF, then INTO activates a type 4 interrupt. The interrupt service routine ends with an IRET instruction that restores control to the instruction following INTO.

CHARACTER TRANSLATION AND STRING INSTRUCTIONS

The instructions in this section operate on characters or string elements instead of logical or numeric values.

Translate (XLAT)

XLAT replaces a byte in the AL register with a byte from a user-coded translation table. When XLAT is performed, AL has the unsigned index to the table addressed by BX. XLAT alters the contents of AL from table index to table entry. BX remains unchanged. The XLAT instruction is helpful for translating from one coding system to another, such as from ASCII to EBCDIC. The translate table can be as long as 256 bytes. The value placed in the AL register is an index to the location of the corresponding translation value. The XLAT instruction can translate a block of codes up to 64K bytes long, when used with a LOOP instruction.

Example: XLAT. Replaces the byte in AL with the byte from the translate table that is specified by the value in AL.

String Manipulation Instructions and Repeat Prefixes

The string instructions (also known as primitives) work on string elements to move, compare, and scan byte or word strings. One-byte repeat prefixes can repeat the operation of a string primitive in order to process strings as long as 64K bytes.

The direction flag (DF) is used by the repeated string primitives to choose left-to-right or right-to-left string processing. The primitives use a count in CX to limit the processing operation. These instructions use the register pair DS:SI to indicate the source string element and the register pair ES:DI to indicate the destination.

Depending on whether the string primitive is operating on byte strings or word strings, one of two possible opcodes represent it. The string primitives are generic and need one or more additional operands to determine the size of the string elements being processed. These operands do not determine the string addresses. Rather, the addresses must already be present in the correct registers.

Each repetition of a string operation using the Repeat prefixes contains the following steps:

1. Respond to pending interrupts.
2. Check CX and stop repeating if it is zero.
3. Execute the string operation once.
4. Modify the memory pointers in DS:SI and ES:DI by incrementing SI and DI if DF is 0 or by decrementing SI and DI if DF is 1.
5. Decrement CX (this step does not affect the flags).
6. For SCAS (Scan String) and CMPS (Compare String), check ZF and stop repeating if it does not match the repeat condition.

The Load String and Store String instructions let a program execute arithmetic or logical operations on string characters, using AX for word strings and AL for byte strings. Repeated operations that contain instructions other than string primitives must use the loop instructions instead of a repeat prefix.

STRING MOVEMENT INSTRUCTIONS

Repeat While CX Not Zero (REP)

REP designates a repeated operation of a string primitive. This prefix invokes the hardware to automatically repeat the associated string primitive until CX = 0. This form of repeat lets the CPU process strings much faster than through the use of a regular software loop.

When the REP prefix is used with a MOVS instruction, it works as a memory-to-memory block transfer. To prepare for this operation, the program must initialize CX and the register pairs DS:SI and ES:DI. The number of bytes or words in the block is specified by CX.

The program must point DS:SI to the first element of the source string and point ES:DI to the destination address for the first element, if DF = 0. However, the program must point these two register pairs to the last element of the source string and to the destination address for the last element, respectively, if DF = 1.

Example: REP MOVSW. The processor checks to see if CX is zero. If CX is not zero, the processor moves a word from the location pointed to by DS:SI to the location pointed to by ES:DI and increments SI and DI by two (if DF = 0). Subsequently, the processor decrements CX by one and begins the repeat cycle again by checking to see if CX is zero. After CX decrements to zero, the processor performs the instruction that follows.

Move String (MOVS)

MOVS transfers the string character pointed to by the combination of DS and SI to the location pointed to by the combination of ES and DI. MOVS is the only memory-to-memory transfer supported by the base architecture's instruction set. MOVSB works on byte elements. Although the destination segment register cannot be overridden by a segment override prefix, the source segment register can be overridden.

Example: MOVSW. Transfers the contents of the memory byte pointed to by DS:SI to the location pointed to by ES:DI.

OTHER STRING OPERATIONS

Compare Strings (CMPS)

CMPS subtracts the destination string element (ES:DI) from the source string element (DS:SI) and updates the flags AF, SF, PF, CF and OF. ZF = 1 if the string elements are equal; otherwise, ZF = 0. The processor increments the memory pointers (SI and DI) for the two strings if DF = 0. Although the destination segment register cannot be overridden by a segment override prefix, the source segment register can be overridden.

Example: CMPSB. Compares the source and destination string elements and returns the result to ZF.

Scan String (SCAS)

SCAS subtracts the destination string element at ES:DI from AX or AL and updates the flags AF, SF, ZF, PF, CF and OF. ZF = 1 if the values are equal; otherwise, ZF = 0. The processor increments the memory pointer (DI) for the string if DF = 0. Although the destination segment register cannot be overridden by a segment override prefix, the source segment register can be overridden.

Example: SCASW. Compares AX with the destination string element.

Repeat While CX Equal/Zero) (REPE/REPZ) and Repeat While CX Not Equal/Not Zero (REPNE/REPNZ)

REPE/REPZ and REPNE/REPNZ are the prefixes used only with the SCAS and CMPS primitives.

The difference between these two prefix bytes is that REPE/REPZ ends when ZF = 0 and REPNE/REPNZ ends when ZF = 1. Initialization before execution of a repeated string instruction is not needed by ZF.

When these prefixes adjust either the SCAS or CMPS primitives, the processor compares the current string element either with AX for word elements or with AL for byte elements. The resulting ZF can then limit the repeated operation and a zero value in CX.

Example: REPE SCASB. Invokes the processor to scan the string pointed to by ES:DI until it encounters a match with the byte value in AL or until CX decrements to zero.

Load String (LODS)

LODS sets the source string element at DS:SI into AX for word strings or into AL for byte strings.

Example: LODSW. Puts the value pointed to by DS:SI into AX.

ADDRESS MANIPULATION INSTRUCTIONS

The set of address manipulation instructions execute address calculations or transfer to a new data segment or extra segment.

Load Effective Address (LEA)

LEA moves the offset of the source operand (instead of its value) to the destination operand. While the source operand must be a memory operand, the destination operand must be a 16-bit general register (AX, DX, BX, CX, BP, SP, SI, or DI).

LEA, which does not affect any flags, is useful for initializing the registers prior to the execution of the string primitives or the XLAT instruction.

Example: LEA BX EBCDIC_TABLE. Causes the processor to set the starting location address of the table labeled EBCDIC_TABLE into BX.

Load Pointer Using DS (LDS)

LDS moves a 32-bit pointer variable from the source operand to DS and the destination register. While the source operand must be a memory operand, the destination operand must be a 16-bit general register (AX, DX, BX, CX, BP, SP, SI or DI). DS takes the high-order segment word of the pointer. The destination register takes the low-order word, which points to a particular location within the segment.

Example: LDS SI, STRING_X. Puts the word identifying the segment pointed to by STRING_X into DS, and puts the offset of STRING_X into SI. A convenient way to prepare for a string operation on a source string that is not in the current data segment is to specify SI as the destination operand.

Load Pointer Using ES (LES)

LES works the same as LDS, except that ES takes the offset word instead of DS.

Example: LES DI, DESTINATION_X. Puts the word identifying the segment pointed to by DESTINATION_X into ES, and puts the offset of DESTINATION_X into DI. This instruction is a convenient way of selecting a destination for a string operation if the desired location is not in the current extra segment.

FLAG CONTROL INSTRUCTIONS

The flag control instructions provide a way to alter the condition of bits in the flag register.

Carry Flag Control Instructions

The carry flag instructions are useful with rotate-with-carry instructions RCL and RCR. Before execution of a rotate that transfers the carry bit into one end of the rotated operand, the carry flag instructions are able to initialize the carry flag, CF, to a known state.

SET CARRY FLAG (STC)

STC changes the carry flag (CF) to 1.

CLEAR CARRY FLAG (CLC)

CLC changes the carry flag (CF) to zero.

COMPLEMENT CARRY FLAG (CMC)

CMC reverses the present status of the carry flag (CF).

Direction Flag Control Instructions

The direction flag control instructions set or clear the direction flag, DF, which controls the left-to-right or right-to-left direction of string processing. The processor automatically increments the string memory pointers, SI and DI, after each execution of a string primitive, if DF = 0. The processor decrements these pointer values, if DF = 1. DF's initial state is 0.

CLEAR DIRECTION FLAG (CLD)

CLD changes DF to 0, causing the string instructions to auto-increment SI and/or DI. No other flags are affected.

SET DIRECTION FLAG (STD)

STD changes DF to 1, causing the string instructions to auto-decrement SI and/or DI. No other flags are affected.

Flag Transfer Instructions

Specific instructions exist to change CF and DF, but there is no direct way to change the other flags. The flag transfer instructions let a program change the other flag bits with the bit manipulation instructions after moving these flags to the stack or the AH register.

The PUSHF and POPF instructions can also preserve the condition of the flag register before performing a procedure.

LOAD AH FROM FLAGS (LAHF)

LAHF copies SF, ZF, AF, PF, and CF to AH bits 7, 6, 4, 2, and 0, respectively (see Figure 3-13). The contents of the remaining bits (5, 3, and 1) are not defined, and the flags remain unaffected. This instruction can help convert 8080/8085 assembly language programs to run on the 8086, 8088, 80186, 80188, and 80C286 base architecture.

STORE AH INTO FLAGS

SAHF moves AH bits 7, 6, 4, 2, and 0 into SF, ZF, AF, PF, and CF, respectively (see Figure 3-13). This instruction is also 8080/8085 compatible with the 8086, 8088, 80186, 80188, and 80C286.

PUSH FLAGS (PUSHF)

PUSHF decrements SP by two and then moves all flags to the word at the top of stack pointed to by SP (see Figure 3-14). The flags are not affected. Using this instruction, a procedure can save the condition of the flag register for later use.

POP FLAGS (POPF)

POPF moves specific bits from the word at the top of stack into the low-order byte of the flag register (see Figure 3-14). SP is then incremented by two.

Note: Unless an application program in the protected virtual address mode is performing at privilege level 0, it may not alter IOPL (the I/O privilege level flag). A program can alter IF (the interrupt flag) only when performing at a level that is at least as privileged as IOPL. This instruction restores the flag status from a prior value.

BINARY-CODED DECIMAL ARITHMETIC INSTRUCTIONS

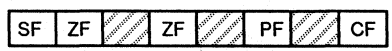
These instructions, which operate only on AL or AH registers, modify the results of a previous arithmetic operation to create a valid packed or unpacked decimal result.

Packed BCD Adjustment Instructions

DECIMAL ADJUST (DAA)

DAA corrects the result of adding two valid packed decimal operands in AL. In order to insure a pair of valid packed decimal digits as results, DAA must always follow the addition of two pairs of packed decimal numbers (one digit in each nibble). If carry was necessary, the carry flag will be set.

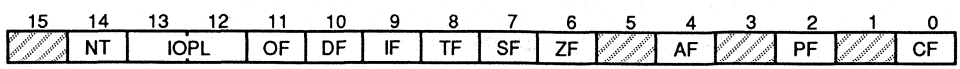
Figure 3-13 LAHF and SAHF



Register AH

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 are indeterminate. 14554A-025

Figure 3-14 PUSHF and POPF



Stack Word

PUSHF decrements SP by 2 bytes (1 word) and copies the contents of the flag register to the top of the stack. POPF loads the flag register with the contents of the last word pushed onto the stack. The bit position of each flag is the same in the stack word as it is in the flag register. Only programs executing at the highest privilege level (level 0) may alter the 2-bit IOPL flag. Only programs executing at a level at least as privileged as that indicated by IOPL may alter IF. 14554A-026

Unpacked BCD Adjustment Instructions

ASCII ADJUST FOR ADDITION (AAA)

AAA converts the contents of register AL to a valid unpacked decimal number, and changes the top 4 bits to zero. AAA must always follow the addition of two unpacked decimal operands in AL. If a carry was necessary, the carry flag will be set and AH will be incremented.

ASCII ADJUST FOR SUBTRACTION (AAS)

AAS converts the contents of register AL to a valid unpacked decimal number, and changes the top 4 bits to zero. AAS must always follow the subtraction of one unpacked decimal operand from another in AL. If a borrow was necessary, the carry flag will be set and AH decremented.

ASCII ADJUST FOR MULTIPLICATION (AAM)

AAM corrects the result of a multiplication of two valid unpacked decimal numbers. In order to secure a valid decimal result, AAM must always follow the product of two decimal numbers. The high-order digit will be in AH, the low-order digit in AL.

ASCII ADJUST FOR DIVISION (AAD)

AAD adjusts 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. The high-order digit will be in AH, the low-order digit in AL. This instruction will modify the value and leave it in AL. AH will be 0.

TRUSTED INSTRUCTIONS

In protected mode operation, the 80C286 processor limits the execution of trusted instructions according to the Current Privilege Level (CPL) and the current value of IOPL, the 2-bit I/O privilege flag. The value of IOPL may be changed only by a program working at the highest privilege level (level 0). A program may perform trusted instructions only when executing at a level at least as privileged as IOPL specifies.

Trusted instructions affect I/O operations, inter-processor communications in a multiprocessor system, interrupt enabling, and the HLT instruction.

These protection considerations are not applicable in the real address mode.

Trusted and Privileged Restrictions on POPF and IRET

POPF and IRET are not affected by IOPL unless they try to change IF (flag register bit 9). POPF must be part of a program that is operating at a privilege level greater than or equal to IOPL's specifications in order to alter IF. Any attempt to change IF when CPL $\geq$ 0 will be overlooked (IF will be ignored). CPL must be 0 to modify the IOPL field.

Machine State Instructions

These trusted instructions alter the machine state control interrupt response, the processor halt state, and the bus LOCK signal that controls memory access in multiprocessor systems.

CLEAR INTERRUPT-ENABLE FLAG (CLI) AND SET INTERRUPT-ENABLE FLAG (STI)

CLI and STI modify bit 9 in the flag register. The processor reacts only to internal interrupts and to non-maskable external interrupts when IF = 0. The processor reacts to all interrupts when IF = 1. An interrupt service routine can use these instructions to bypass further interruption while processing a previous interrupt request. The processor clears IF during initialization, as with the other flag bits. These instructions may be performed only if CPL ≤ IOPL. If they are executed when CPL > IOPL, a protection exception will result.

Example: STI. Sets IF = 1, which facilitates the processing of maskable external interrupts.
Example: CLI. Sets IF = 0, which prevents maskable interrupt processing.

HALT (HLT)

HLT invokes the processor to delay processing operations pending an interrupt or a system reset. This trusted instruction supplies an alternative to an endless software loop in situations where a program must wait for an interrupt. The return address saved after the interrupt will point to the instruction just after HLT. This instruction may be performed only when CPL = 0.

ASSERT BUS LOCK (LOCK)

LOCK is a *1-byte* prefix code that causes the processor to signal bus LOCK during execution of the subsequent instruction. No flags are affected, and LOCK may be used only when CPL ≤ IOPL. If LOCK is used when CPL > IOPL, a protection exception will result.

Input and Output Instructions

These trusted instructions give access to the processor's I/O ports to move data to and from peripheral devices. In protected mode, these instructions may be performed only when CPL ≤ IOPL.

INPUT FROM PORT (IN)

IN moves a byte or a word from an input port to AL or AX. The processor moves 8 bits from the selected port to AL if a program specifies AL with the IN instruction. Alternately, the processor moves 16 bits from the port to AX if a program specifies AX with the IN instruction.

The program has two ways to specify the port's number. With an immediate byte constant, the program specifies 256 8-bit ports numbered 0 through 255 or 128 16-bit ports numbered 0,2,4, ... 252,254. Using the present value in DX, the program specifies 8-bit ports numbered 0 through 65,535 or even-numbered 16-bit ports in the same range.

Example: IN AL, BYTE_PORT_NUMBER.
Moves 8 bits from the port identified by the immediate constant BYTE_PORT_NUMBER to AL.

OUTPUT TO PORT (OUT)

OUT moves a byte or a word to an output port from AL or AX. The program can specify the port's number using the identical methods of the IN instruction.

Example: OUT AX, DX. Moves 16 bits from AX to the port identified by the 16-bit number in DX.

INPUT STRING (INS) AND OUTPUT STRING (OUTS)

INS and OUTS invoke block input or output operations using a Repeat prefix. Chapter 4 contains more information on INS and OUTS.

PROCESSOR EXTENSION INSTRUCTIONS

Processor Extension instructions extend the base architecture of the CPU, an axample of which is the AMD 80C287 math coprocessor. In order to support high-precision integer and floating-point calculations, the AMD 80C287 math coprocessor expands the instruction set of the CPU-based architecture to include arithmetic, comparison, transcendental, and data transfer instructions. The AMD 80C287 math coprocessor also has a set of constants to improve the speed of numeric calculations.

A program includes instructions for the AMD 80C287 math coprocessor in logical order with the instructions for the CPU. These instructions are performed by the system in the same order as they appear in the instruction stream. The AMD 80C287 device operates together with the CPU to supply maximum throughput for numeric calculations.

The software emulation of the AMD 80C287 math coprocessor is transparent to application software, but needs additional time for execution.

Processor Extension Synchronization Instructions

Escape and wait instructions let a processor extension like the AMD 80C287 device acquire instructions and data from the system bus and wait for the AMD 80C287 math coprocessor to return a result.

ESCAPE (ESC)

ESC designates floating-point numeric instructions and lets the 80C286 send the opcode to the AMD 80C287 math coprocessor or move a memory operand to the AMD 80C287 device. With the Escape instructions, the AMD 80C287 math coprocessor can perform high-performance, high-precision floating-point arithmetic that complies with the IEEE floating-point standard 754.

Example: ESC 6, ARRAY [SI]. The CPU sends to the AMD 80C287 math coprocessor the escape opcode 6 and the location of the array pointed to by SI.

WAIT

WAIT delays program execution until the 80C286 CPU detects a signal on the $\overline{\text{BUSY}}$ pin. In a configuration that contains a math processor extension, the AMD 80C287 device invokes the $\overline{\text{BUSY}}$ pin to signal that it has finished its processing task and that the CPU may acquire the results.

Numeric Data Processor Instructions

This section discusses the categories of instructions included in AMD 80C286 systems that contain an AMD 80C287 math coprocessor or a software emulation of this processor
extension.

ARITHMETIC INSTRUCTIONS

The expanded instruction set contains the four arithmetic operations (add, subtract, multiply, and divide), and subtract-reversed and divide-reversed instructions. The

arithmetic functions contain square root, modulus, absolute value, integer part, change sign, scale exponent, and extract exponent instructions.

COMPARISON INSTRUCTIONS

The comparison instructions consist of the compare, examine, and test instructions. Special forms of the compare instruction can enhance algorithms by allowing comparisons between binary integers and real numbers in memory.

TRANSCENDENTAL INSTRUCTIONS

These instructions execute the calculations for all common trigonometric, inverse trigonometric, hyperbolic, inverse hyperbolic, logarithmic, and exponential functions. The transcendental instructions include tangent, arctangent, $2x-1$, $Y.\log_2 X$, and $Y.\log_2 (X+1)$.

DATA TRANSFER INSTRUCTIONS

The data transfer instructions shift operands among the registers and between a register and memory. This group contains the load, store, and exchange instructions.

CONSTANT INSTRUCTIONS

Each of the constant instructions puts a frequently used constant into an AMD 80C287 register. The values, with a real precision of 64 bits, are correct to approximately 19 decimal places. The constants placed by these instructions include 0, 1, Pi, $\log_e 10$, $\log_2 e$, $\log_{10} 2$, and $\log 2_e$.

CHAPTER 4

EXTENDED INSTRUCTION SET

The instructions discussed in this chapter expand the capabilities of the base architecture instruction set described in Chapter 3. These extensions include new instructions and variations of some instructions that are not strictly part of the base architecture (i.e., not part of the 8086 and 8088). These instructions are also possible on the 80186 and 80188. The instruction variations discussed in Chapter 3 contain the immediate forms of the PUSH and MUL instructions, PUSHA, POPA, and the privilege level restrictions on POPF.

The string input and output instructions (INS and OUTS), the ENTER procedure and LEAVE procedure instructions, and the check index BOUND instruction are among the new instructions covered in this chapter.

BLOCK I/O INSTRUCTIONS

The repeat prefix (REP), adjusts the INS and OUTS instructions to supply a way of moving blocks of data between an I/O port and memory. These block I/O instructions are string primitives that simplify programming and improve the speed of data transfer by eliminating the need to use a separate LOOP instruction or an intermediate register to hold the data.

To use trusted instructions such as INS and OUTS, a program must perform at a privilege level at least as privileged as that specified by the 2-bit IOPL flag (CPL ≤ IOPL). A protection exception will occur if there is any attempt by a less-privileged program to use a trusted instruction.

Each string primitive, depending on whether it operates on byte strings or word strings, is represented by one of two possible opcodes. The memory address in SI or DI is updated by 1 for byte values and by 2 for word values following each transfer. Depending on the value in the DF field, SI or DI is auto incremented (DF = 0) or auto-decremented (DF = 1).

INS and OUTS utilize DX to specify I/O ports numbered 0 through 65,535 or 16-bit ports in the same range using only even port addresses.

INS moves a byte or a word string element from an input port to memory. The processor moves 8 bits from the selected port to the memory location ES:DI suggests, if a program specifies INSB. On the other hand, the processor moves 16 bits from the port to the memory location ES:DI suggests, if a program specifies INSW. The destination segment register choice (ES) cannot be modified for the INS instruction.

Together with the REP prefix, INS transfers a block of information from an input port to a series of consecutive memory locations.

Example: REP INSB. The processor repeatedly moves 8 bits to the memory location suggested by ES:DI from the port chosen by the 16-bit port number in DX. The CPU decrements CX after each byte transfer, and the instruction ends the block transfer when CX = 0. The processor increments DI by one if DF = 0 after decrementing CX. If DF = 1, it decrements DI by one.

OUTS moves a byte or a word string element to an output port from memory. Together with the REP prefix, OUTS transfers a block of information from a series of consecutive memory locations suggested by DS:SI to an output port.

Example: REP OUTS WSTRING. Assuming that the program proclaims WSTRING to be a word-length string element, the assembler takes the 16-bit form of the OUTS instruction and creates the object code for the program. The processor repeatedly moves words from the memory locations suggested by DI to the output port chosen by the 16-bit port number in DX.

The CPU decrements CX after each word transfer, and the instruction ends the block transfer when CX = 0. The processor increments SI by two to point to the subsequent word in memory if DF = 0 after decrementing CX. If DF = 1, it decrements SI by two.

HIGH-LEVEL INSTRUCTIONS

These instructions furnish machine-language functions typically present only in high-level languages. ENTER and LEAVE are two of the instructions that simplify procedure programming. BOUND is an easy way to test an index against its already defined range.

Enter Procedure (ENTER) produces the stack frame needed by most block-structured high-level languages. A LEAVE instruction at the end of a procedure balances an ENTER at the start of the procedure to ease stack management and manage access to variables for nested procedures.

Example: ENTER 2048,3. Appropriates 2048 bytes of dynamic storage on the stack and sets up pointers to two previous stack frames that ENTER produces for this procedure.

The ENTER instruction is composed of two parameters. One indicates the number of bytes of dynamic storage to be appropriated on the stack for the routine being entered. The second parameter conforms to the lexical nesting level (0–31) of the routine. (Note: The lexical level has no relationship to the protection privilege levels or the I/O privilege level.)

The specified lexical level establishes the number of sets of stack frame pointers the CPU duplicates into the new stack frame from the previous frame. This list of stack frame pointers is also known as the display. The display's first word is a pointer to the last stack frame. With this pointer, a LEAVE instruction can reverse the action of the preceding ENTER instruction by throwing away the last stack frame.

After ENTER produces the procedure's new display, the instruction appropriates the dynamic storage space for that procedure by reducing SP by the number of bytes indicated in the first parameter. SP's new value becomes a base for all PUSH and POP operations within that procedure.

ENTER leaves BP pointing to the start of the new stack frame to allow a procedure to address its display. Data manipulation instructions labeling BP as a base register implicitly address locations within the stack segment rather than the data segment. Nested and non-nested are the two forms of the ENTER instruction. The non-nested form is used when the lexical level equals 0. Because the second operand equals 0, ENTER pushes BP, duplicates SP onto BP, and then subtracts the first operand from SP. When the second parameter (lexical level) is not equal to 0, the nested form of ENTER is used.

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 BP
Set a temporary value FRAME_PTR: = SP
If LEVEL > 0 then
        Repeat (LEVEL –1) times:
            BP: = BP –2
            Push the word pointed to by BP
        End repeat
        Push FRAME_PTR
End if
BP: = FRAME_PTR
SP: = SP–first operand.
```

While the other procedures are nested within, the main procedure functions at level 1, the highest lexical level. The first procedure it calls functions at level 2, the next deeper lexical level. The variables of the main program, which are at set locations determined by the compiler, can be accessed by a level 2 procedure. In a level 1 procedure, ENTER appropriates only the desired dynamic storage on the stack because no prior display exists to be copied.

It is necessary for a program running at a higher lexical level calling a program at a lower lexical level that the variables of the calling program be accessible to the called procedure. ENTER supplies this access through a display with the ability to address the calling program's stack frame.

A procedure calling another procedure at the same lexical level suggests that they are parallel procedures and that the variables of the calling procedure are not accessible to the called procedure. Under these conditions, ENTER duplicates only that part of the display from the calling procedure that references previously nested procedures running at higher lexical levels. The pointer for addressing the calling procedure's stack frame is not part of the new stack frame.

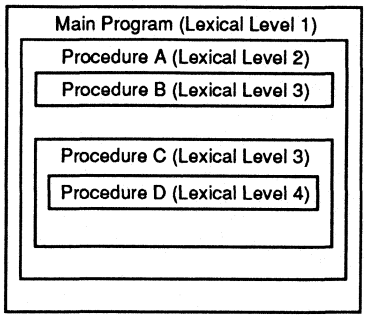
ENTER regards a reentry procedure as a procedure calling another procedure at the same lexical level. In this situation, each subsequent repeat of the reentry procedure can address only its own variables and the variables of the calling procedures at higher lexical levels. A reentry procedure, however, does not need pointers to the stack frames of preceding repeats. It may continually address its own variables.

By duplicating just the stack frame pointers of procedures at higher lexical levels, the ENTER instruction ensures that procedures access only those variables of higher lexical levels and not those at parallel lexical levels (see Figure 4-1). Figures 4-1a, 4-1b, 4-1c, and 4-1d illustrate the conduct of the ENTER instruction if the modules in the definition of ENTER were to call each other alphabetically.

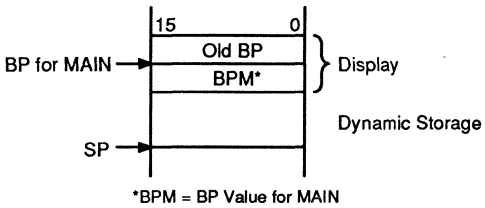
Block-structured high-level languages can apply the lexical levels, as denoted by ENTER, to oversee access to the variables of formerly nested procedures. As a model, if PROCEDURE A calls PROCEDURE B, which, in turn, calls PROCEDURE C, then the variables of MAIN and PROCEDURE A are accessible to PROCEDURE C, but not PROCEDURE B, because they function at the same lexical level. A complete definition of the variable access for Figure 4-1 appears below.

1. MAIN PROGRAM contains variables at fixed locations.
2. Only the fixed variables of MAIN are accessible to PROCEDURE A.

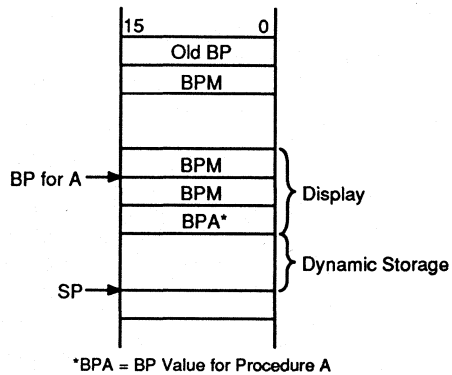
Figure 4-1 Variable Access in Nested Procedures



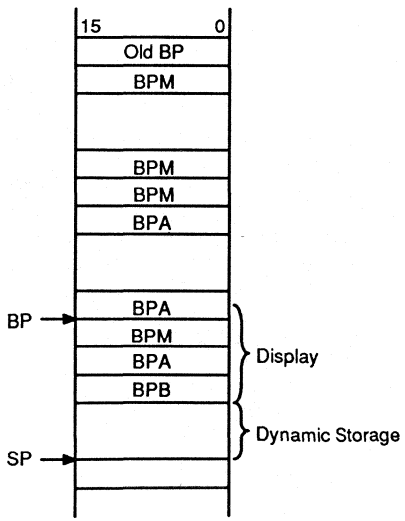
a. Stack Frame for MAIN at Level 1



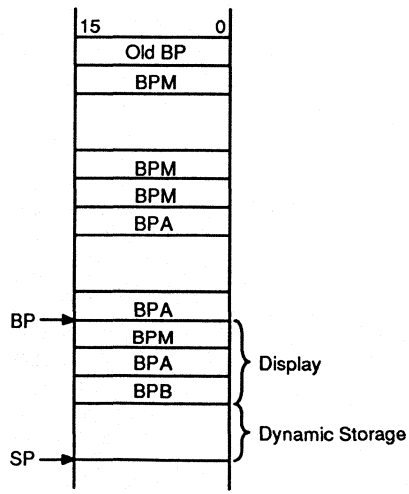
b. Stack Frame for Procedure A



c. Stack Frame for Procedure B at Level 3 Called from A



d. Stack Frame for Procedure C at Level 3 Called from B



14554A-027

3. Only the variables of PROCEDURE A and MAIN are accessible to PROCEDURE B. The variables of PROCEDURE C or PROCEDURE D are inaccessible to PROCEDURE B.
4. Only the variables of PROCEDURE A and MAIN are accessible to PROCEDURE C. The variables of PROCEDURE B or PROCEDURE D are inaccessible to PROCEDURE C.
5. The variables of PROCEDURE C, PROCEDURE A, and MAIN are accessible to PROCEDURE D. The variables of PROCEDURE B are inaccessible to PROCEDURE D.

ENTER at the start of the MAIN PROGRAM produces dynamic storage space for MAIN, but duplicates no pointers. The only word in the display points to itself because no prior value exists for LEAVE to return to BP (see Figure 4-1a).

ENTER produces a new display for PROCEDURE A after MAIN calls PROCEDURE A. The first word in the new display points to the prior value of BP (BPM for LEAVE to return to the MAIN stack frame), and the second word points to BP's present value. Because MAIN is at level 1, PROCEDURE A can access variables in MAIN. Thus, the dynamic storage's base for MAIN is at [BP – 2]. Each dynamic variable for MAIN will be at a predetermined offset from this value (see Figure 4-1b).

ENTER creates a new display for PROCEDURE B after PROCEDURE A calls PROCEDURE B. The first word of the new display points to the prior value of BP, the second word points to BP's value for MAIN, the third word points to BP's value for A, and the last word points to the present BP. B can access variables in A and MAIN by obtaining the base addresses of the respective dynamic storage areas from the display (see Figure 4-1c).

ENTER creates a new display for PROCEDURE C after PROCEDURE B calls PROCEDURE C. The first word of the display points to the prior value of BP, the second word points to BP's value for MAIN, the third word points to BP's value for A, and the last word points to the present value of BP. Since PROCEDURE B and PROCEDURE C have the same lexical level, PROCEDURE C cannot access variables in B. Thus, PROCEDURE C does not receive a pointer to the start of PROCEDURE B's stack frame (see Figure 4-1d).

Leave Procedure (LEAVE) reverses the prior ENTER instruction's action. No operands are included in the LEAVE instruction.

Example: LEAVE. LEAVE first duplicates BP to SP to expel all stack space appropriated to the procedure by the most recent ENTER instruction. LEAVE then pops BP's old value from the stack. A RET instruction, which can remove any arguments pushed onto the stack by the calling program for the called procedure's use, follows.

Detect Value Out of Range (BOUND) confirms that the signed value in the identified register is within specified limits. If the value in the register is less than the lower bound or greater than the upper bound, an interrupt (INT 5) results.

Two operands are contained in the BOUND instruction. The first operand indicates the register being tested, and the second includes the effective relative address of the two signed BOUND limit values. The BOUND instruction presumes that it can get the upper limit from the memory word that immediately succeeds the lower limit. An invalid opcode exception results if these limit values are used as register operands.

BOUND can check array bounds before a new index value is used, in order to access an element within the array. BOUND supplies an easy way to check an index register's value before the program overwrites information somewhere outside the array's limits.

The two-word memory block that identifies the lower and upper limits of an array might generally reside just prior to the array itself. Thus, the array bounds are accessible at a constant offset of −4 from the start of the array. Because the address of the array will already be listed in a register, additional calculations to get the array bounds' effective address are avoided.

Example: BOUND BX,ARRAY−4. Relates the value in BX to the lower limit at address ARRAY−4 and the upper limit at address ARRAY−2. The interrupt for this instruction (INT 5) results if the signed value in BX is less than the lower bound or greater than the upper bound. This instruction has no effect in other cases.

CHAPTER 5

REAL ADDRESS MODE

According to the status of the Protection Enabled bit of the MSW status register, the 80C286 can be operated in one of two modes. The modes and mode bits of the 80C286 differ from those of some processors because they do not represent a radical transition between opposing architectures. Rather, the Protection Enabled bit's setting just determines whether particular advanced features, besides the baseline architecture of the 80C286, will be available to system designers and programmers.

The processor switches to Protected Virtual Address mode if the programmer sets the Protection Enabled (PE) bit. Memory addressing is executed in terms of virtual addresses, in this mode of operation, with on-chip mapping mechanisms executing the virtual-to-physical transition. Only in this mode can the system designer use the 80C286's advanced architectural features, which include new features such as virtual memory support, system-wide protection, and built-in multi-tasking mechanisms.

Initially, the processor begins in Real Address mode when the system is reset. All memory addressing is executed in terms of real physical addresses in this mode of operation. Thus, the architecture of the 80C286 in real address mode is the same as the 8086 and other processors in the 8086 family.

ADDRESSING AND SEGMENTATION

When operated in real address mode, the 80C286 supplies a one-megabyte memory space (2^{20} bytes), like other processors in the 8086 family. Physical addresses, the 20-bit values that uniquely specify each byte location in this address space, range from 0 through FFFFFH. Address bits A23–20, which are not always zero in real address mode, should not be used by the system while the processor is in real address mode.

An address is identified by a 32-bit pointer that has two components:

1. A 16-bit effective address offset that establishes the displacement, in bytes, of a certain location in a segment;
2. A 16-bit segment selector component that establishes the segment's starting address.

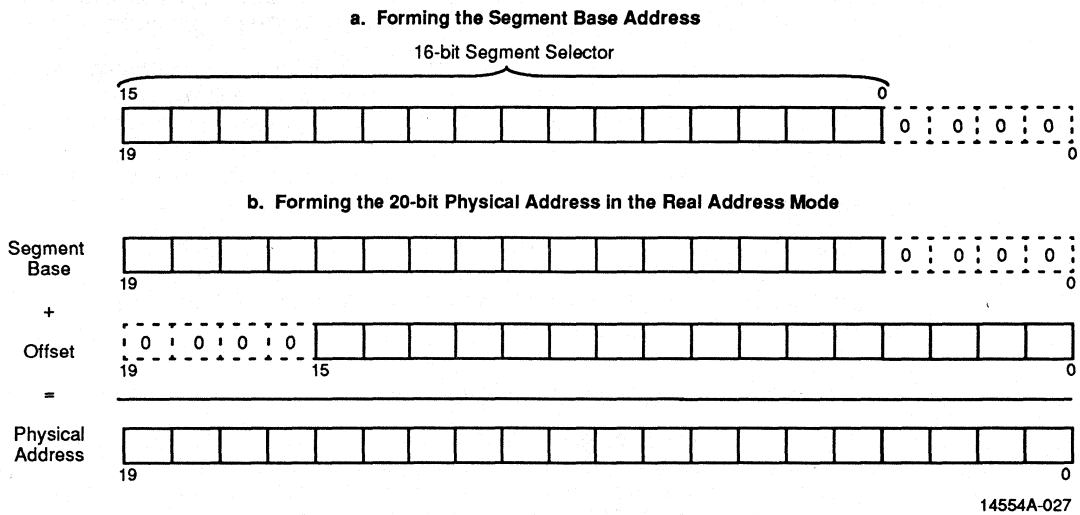
An instruction (such as JMP, LES, LDS, or CALL) can explicitly reference both address components. However, the segment selector is usually the contents of a segment register.

The effective-address offset's interpretation is clear. Segments are 64K (2^{16}) bytes long at the most. Therefore, an unsigned 16-bit quantity can address any arbitrary byte location with a segment. The offset of the lowest-addressed byte in a segment is 0, and the offset of the highest-addressed byte is FFFFH. Data operands must be contiguous and contained completely in a segment. (The above rules are applicable in both modes.)

The second component of a logical address is a segment selector, which identifies the starting address of a segment within a physical address space of 2^{20} bytes. The segment selector is a 16-bit quantity.

The 80C286 creates a 20-bit physical address from a segment selector and offset value when accessing memory in real address mode. The segment selector value, shifted left four bit positions, forms the segment base address. The offset expands by 4 high-order zeroes and is then added to the base to create the physical address (see Figure 5-1).

Figure 5-1 Forming Addresses in the Read Address Mode



Therefore, each segment must begin at a byte address that can be divided evenly by 16. Because of this, each segment is set at a 20-bit physical address whose least significant four bits are zeroes. This set-up lets the 80C286 interpret a segment selector as the high-order 16 bits of a 20-bit segment base address.

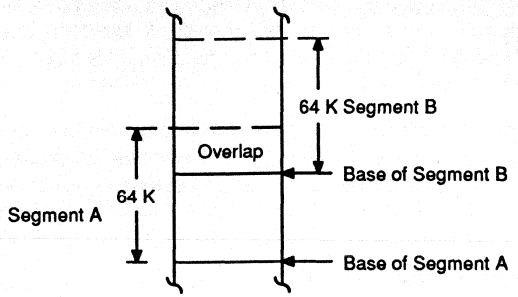
The 80C286 does not perform any limit or access checks in the real address mode. Rather, all segments are readable, writable, executable, and limited to 0FFFFH (65,535 bytes). Unused portions of a segment can be used as another segment by overlapping the two (see Figure 5-2), in order to save physical memory. The segment override and group operators enable the 8086 software development tools to support this feature.

Note: Programs that access segment B from segment A , however, are no longer compatible in the protected virtual address mode.

INTERRUPT HANDLING

Program interrupts can be produced in one of two separate ways. The currently executing program directly causes an internal interrupt. An interrupt occurs when a particular instruction is performed, whether on purpose (an INT n instruction) or as an unanticipated exception (invalid opcode). Alternately, an event external to the processor can cause an external interrupt to occur asynchronously, and this interrupt does

Figure 5-2 **Overlapping Segments to Save Physical Memory**



14554A-027

not necessarily have a relationship with the currently executing program. The INTR and NMI pins of the 80C286 supply a way for external hardware to signal that such events have occurred.

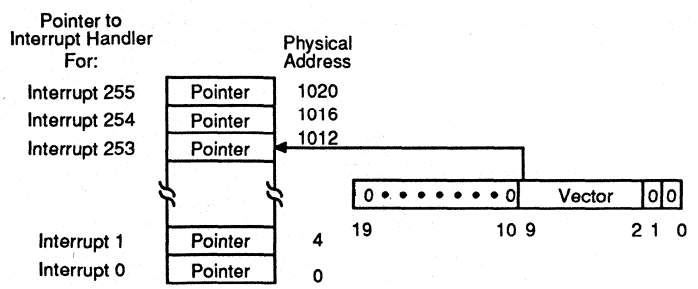
Interrupt Vector Table

Whether its origin is internal or external, an interrupt commands instant attention from an associated service routine. Control must be shifted from the presently executing program to the appropriate interrupt service routine, at least for the moment. Through interrupt vectors, the 80C286 manages such control transfers similarly for both kinds of interrupts.

An interrupt vector, an unsigned integer, ranges from 0–255. Such a vector is assigned to each interrupt. Sometimes, the assignment is already made and fixed. An external NMI interrupt, for example, is always associated with vector 2, while vector 0 is always associated with an internal divide exception. In most instances, though, this association between an interrupt and a vector is determined dynamically. An external INTR interrupt, for instance, provides a vector when responding to an interrupt acknowledge bus cycle. The INT n instruction, on the other hand, provides a vector incorporated within the actual instruction. The vector shifts two places to the left to create a byte address into the table. (See Figure 5-3.)

Whatever the case, the 80C286 uses the interrupt vector as an index into a table to determine the corresponding interrupt service routine's address. This table is called the Interrupt Vector Table for real address mode, and its format is shown in Figure 5-3.

Figure 5-3 **Interrupt Vector Table for Real Address Mode**



14554A-028

The Interrupt Vector Table can have up to 256 consecutive entries. Each entry, four bytes in length, specifies the service routine's address to be associated with the correspondingly numbered interrupt vector code. A full 32-bit pointer, including a 16-bit offset and a 16-bit segment selector, specifies an address within each entry. Interrupts 0–31 are reserved.

The interrupt table is directly accessible at physical memory location 0 through 1023 in Real Address mode. However, the interrupt vector table in the protected Virtual Address mode has no fixed physical address and cannot be directly accessed. Thus, Real Address mode programs that directly control the interrupt vector table will not function in the Protected Virtual Address mode.

INTERRUPT PRIORITIES

Concurrent interrupt requests are processed in a fixed order shown in Table 5-1. Saving the flags, the return address, and setting CS:IP to point at the interrupt handler's first instruction are all factors in interrupt processing. Other interrupts, if still enabled, are processed prior to the execution of the current interrupt handler's first instruction. Therefore, the final interrupt processed is the first one serviced.

Interrupt Procedures

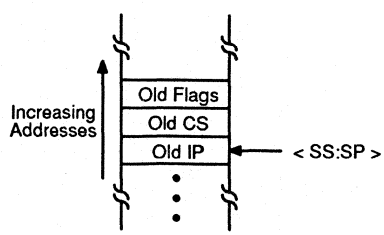
The 8086 executes the following series of steps when an interrupt occurs in real address mode. In the first step, the FLAGS register, along with the old values of CS and IP, is pushed onto the stack (see Figure 5-4), and the IF and TF flag bits are cleared. Next, the vector number reads the interrupt service routine's address from the interrupt table. This address is where execution begins.

When control is transferred to an interrupt service routine, the return linkage is put on the stack, interrupts are disabled, and single-step trace (if in effect) is turned off. At the end of the interrupt service routine, the steps are reversed by the IRET instruction, before shifting control to the program that was interrupted.

Table 5-1 **Interrupt Processing Order**

Order	Interrupt
1.	Instruction exception
2.	Single-step
3.	NMI
4.	Processor extension segment overrun
5.	INTR

Figure 5-4 **Stack Structure after Interrupt (Real Address Mode)**



14554A-029

Registers besides other IP, CS, and FLAGS can be affected by an interrupt service routine. It is the interrupt routine's responsibility to save additional context information before continuing, so that the machine's condition can be restored when the interrupt service routine is finished. (PUSHA and POPA instructions are provided for such operations.) Lastly, performance of the IRET instruction pops the old IP, CS, and FLAGS from the stack and continues with the interrupted program's execution.

Reserved and Dedicated Interrupt Vectors

Basically, the system designer may use almost any interrupt vectors for any reason. However, some of the lowest-numbered vectors are retained for dedicated functions, and their use is specifically indicated by certain kinds of exceptions. Of the first 32 vectors, not one should be defined by the user. Rather, these vectors are either activated by predefined exceptions or reserved for further expansion. The 80C286's dedicated and reserved vectors in real address mode are illustrated in Table 5-2.

Table 5-2 Dedicated and Reserved Interrupt Vectors in Real Address Mode

Function	Interrupt Number	Related Instructions	Return Address Before Instruction Causing Exception?
Divide error exception	0	DIV, IDIV	Yes
Single step interrupt	1	All	N/A
NMI interrupt	2	All	N/A
Breakpoint interrupt	3	INT	N/A
INTO detected overflow exception	4	INTO	No
BOUND range exceeded exception	5	BOUND	Yes
Invalid opcode exception	6	Any undefined opcode	Yes
Processor extension not available exception	7	ESC or WAIT	Yes
Interrupt table limit too small	8	LIDT	Yes
Processor extension segment overrun interrupt	9	ESC	Yes
Segment overrun exception	13	Any memory reference instruction that attempts to reference 16-bit word at offset OFFFFH.	Yes
Reserved	10–12, 14, 15		
Processor extension error interrupt	16	ESC or WAIT	N/A
Reserved	17–31		
User defined	32–255		

The dedicated interrupt vectors' purpose and function is outlined below (the CS:IP's saved value contains all leading prefixes):

- *Divide error (Interrupt 0).* This exception results if the quotient is too great or an attempt is made to divide by zero utilizing the DIV or IDIV instruction. The saved CS:IP points at the failing instruction's primary byte. DX and AX are not changed.
- *Single-Step (Interrupt 1).* If the Trap Flag (TF) bit of the FLAGS register is fixed, this interrupt results following each instruction. To avoid infinite recursion, TF is cleared upon entry to this or any other interrupt. The CS:IP's saved value points to the subsequent instruction.

- *Nonmaskable (Interrupt 2).* Upon receipt of an external signal on the NMI pin, this interrupt results. Ordinarily, the nonmaskable interrupt imposes power-fail/ auto-restart procedures. The CS:IP's saved value points to the interrupted instruction's first byte.
- *Breakpoint (Interrupt 3).* This interrupt results from performance of the one-byte breakpoint instruction. Helpful in the implementation of software debuggers, this instruction requires just one code byte and can replace any instruction opcode byte. The CS:IP's saved value points to the subsequent instruction.
- *INTO Detected Overflow (Interrupt 4).* Performance of the INTO conditional software interrupt instruction brings about this interrupt if the overflow bit (OF) of the FLAGS register is set. The CS:IP's saved value points to the subsequent instruction.
- *BOUND Range Exceeded (Interrupt 5).* This interrupt results upon performance of the BOUND instruction if the identified array index is declared invalid in regard to the stated array bounds. The CS:IP's saved value points to the BOUND instruction's first byte.
- *Invalid Opcode (Interrupt 6).* This exception results when performance of an invalid opcode is attempted. (The majority of the protected virtual address mode instructions in real address mode are classified as invalid and should not be implemented.) If the effective address stated by certain instructions, mainly BOUND, LDS, LES, and LIDT, identifies a register instead of a memory location, this exception also occurs. The saved CS:IP points to the invalid instruction or opcode's first byte.
- *Processor Extension Not Available (Interrupt 7).* Upon performance of the ESC instruction, this interrupt will occur if the status bits of the MSW show that processor extension functions are to be emulated in software. The saved CS:IP points to the ESC or WAIT instruction's first byte.
- *Interrupt Table Limit Too Small (Interrupt 8).* If the limit of the interrupt vector table was altered from 3FFH by the LIDT instruction and an interrupt whose vector is outside the limit, this interrupt will result. The saved CS:IP points to the first byte of the instruction that resulted in the interrupt or that was ready to perform prior to the occurrence of an external interrupt. No error code is pushed.
- *Processor Extension Segment Overrun (Interrupt 9).* If a processor extension memory operand does not fit in a segment, this interrupt results. The saved value of CS:IP points at the first byte of the instruction that resulted in the interrupt.
- *Segment Overrun Exception (Interrupt 13).* This interrupt results when a memory operand cannot fit in a segment. In Real Mode, this occurs only when a word operand starts at segment offset 0FFFFH. The saved value of CS:IP points at the first byte of the instruction that brought about the interrupt. No error code is pushed.
- *Processor Extension Error (Interrupt 16).* This interrupt appears following the numeric instruction that caused the error, and can only occur while performing a subsequent WAIT or ESC. The saved CS:IP points to the ESC or WAIT instruction's first byte. The failed numeric instruction's address is saved in the AMD 80C287 math coprocessor.

SYSTEM INITIALIZATION

An efficient way to start or restart an executing system is supplied by the 80C286. Specific processor registers go into the determinate state upon receipt of the RESET signal (see Table 5-3).

Table 5-3 **Processor State after RESET**

Register	Contents
FLAGS	0002 (H)
MSW	FFF0 (H)
IP	FFF0 (H)
CS	F000 (H)
DS	0000 (H)
SS	0000 (H)
ES	0000 (H)

The processor performs its first instruction at physical address FFFF0H, since the CS register includes F000 (identifying a code segment that begins at physical address F0000) and the instruction pointer includes FFF0. Thus, the uppermost 16 bytes of physical memory are maintained for initial startup logic. Typically, this location includes an intersegment direct JMP instruction whose target is the start of a system initialization or restart program itself.

Some of the steps usually executed by a system initialization routine are listed below:

- Appropriate a stack.
- From secondary storage, load programs and data into memory.
- Initialize external devices.
- Facilitate interrupts (set the IF bit of the FLAGS register), and then set any other desired FLAGS bit.
- If a coprocessor is present or if software is to emulate coprocessor functions, set the applicable MSW flags.
- As applicable, set other registers to the desired initial values.
- Execute. (This last step is usually completed as an intersegment JMP to the main system program.)

CHAPTER 6

MEMORY MANAGEMENT AND VIRTUAL ADDRESSING

The 80C286 supplies an advanced architecture that maintains substantial compatibility with the 8086 and other processors in the 8086 family in protected virtual address mode. The processor's baseline architecture remains constant, in most cases, whatever the mode of operation. As in real address mode, application programmers still use the identical instruction set, addressing modes, and data types in protected mode.

The fact that the protected mode equips system programmers with additional architectural features, supplemental to the baseline architecture, to be taken advantage of in the design and implementation of advanced systems, is the main difference between the two modes. The mechanisms supplied for memory management, protection, and multi-tasking are most noteworthy.

This chapter, which is concerned with protected mode's memory management mechanisms, further discusses the concept of a virtual address and the process of virtual-to-physical address translation. The following chapters examine other key aspects of protected mode operation.

MEMORY MANAGEMENT OVERVIEW

A memory management scheme interjects a mapping operation between logical addresses (addresses as seen by programs) and physical addresses (actual addresses in real memory). The mapping, or assignment of real address space to virtual address space, is transparent to software, since the logical address spaces are unrelated to physical memory (dynamically able to relocate). This lets the program development tools for static systems or the system software for reprogrammable systems monitor the allocation of space in real memory, regardless of individual programs' specifics.

Since application programs work strictly with virtual addresses, they can be translated and loaded independently. In order to use any segments of physical memory available, any program can be relocated.

When operated in protected mode, the 80C286 supplies an effective on-chip memory management architecture. Furthermore, the 80C286 maintains the implementation of virtual memory systems. In other words, the 80C286 supports the implementation of systems that dynamically trade chunks of code and data between real memory and secondary storage devices (a disk) independent of and transparent to the performing application programs. As a result, a program-visible address should be called a virtual address instead of a logical address, since it may allude to a location that does not currently exist in real memory.

Thus, memory management contains a mechanism for mapping the virtual addresses that the program can identify onto the physical addresses of real memory. Segmentation is the basis of virtual memory addressing with the 80C286. Virtual memory is divided into multiple individual segments. These segments are the units of memory that are mapped into physical memory and then traded to and from secondary stor-

age devices. The majority of this chapter is concerned with a detailed description of the 80C286's mapping and virtual memory mechanisms.

Because distinct memory mappings may be allocated to the various tasks in a multi-task or multi-user environment, the idea of a task plays an important role in memory management. For now, it is enough to consider a task an ongoing process, or execution path, that is reserved for a specific function. For instance, in a multi-user time-sharing environment, the processing needed to communicate with a certain user may be regarded as one task, functionally independent of the system's other tasks or users.

VIRTUAL ADDRESSES

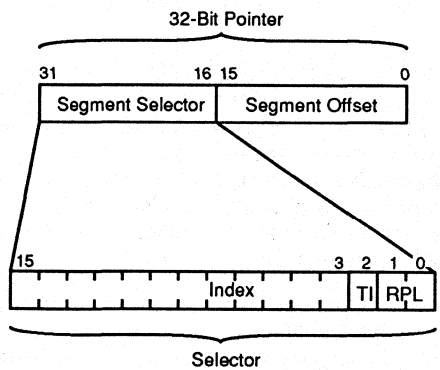
Application programs in protected mode deal only with virtual addresses, and programs are not allowed access to the actual physical addresses produced by the processor. As outlined by Chapter 2, a program identifies an address in terms of two components: 1) a 16-bit effective address offset that specify the displacement, in bytes, of a location within a segment, and 2) a 16-bit segment selector that distinctively references a certain segment. Together, these components comprise a complete 32-bit address (pointer data type), as illustrated by Figure 6-1.

Programs control these 32-bit virtual addresses just like the two-component addresses of real address mode. After a program takes the segment selector component of an address and loads it into a segment register, each succeeding reference to locations within the selected segment must have only a 16-bit offset identified. Location of reference usually guarantees that addresses can be identified efficiently, using just 16-bit offsets.

However, a significant difference between real address mode and protected mode involves the actual format and information content of segment selectors. As with the 8086 and other processors in its family, a 16-bit selector is simply the upper bits of a segment's physical base address in real address mode. Segment selectors in protected mode, however, are a completely different format. (See Figure 6-1.)

Two of the selector bits, indicated by the RPL field in Figure 6-1, are not actually concerned with the selection and specification of segments.

Figure 6-1 **Format of the Segment Selector Component**



14554A-029

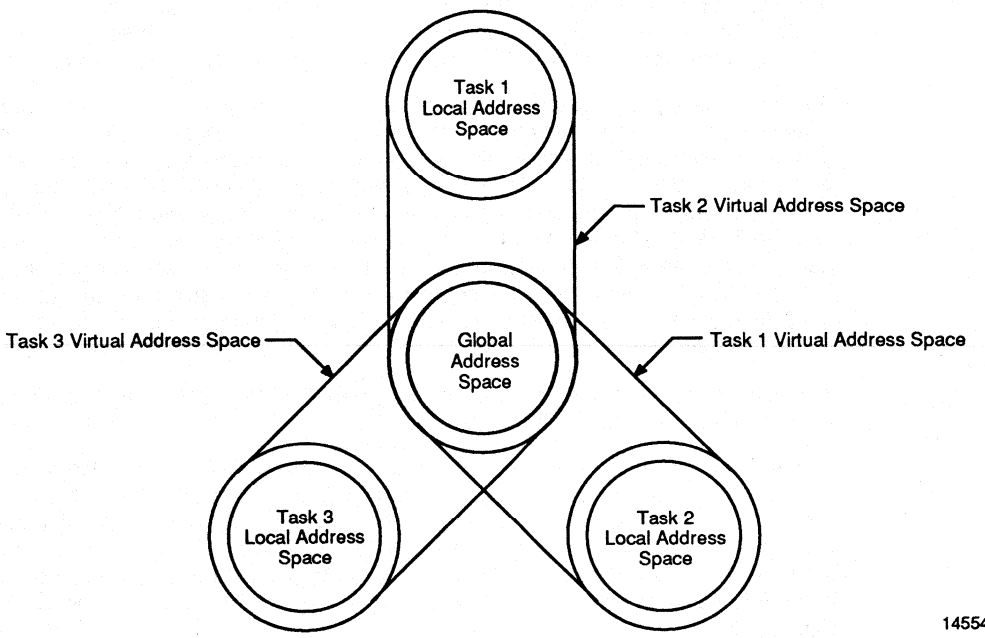
The remaining14 bits of the selector component uniquely label a certain segment. Therefore, the virtual address space of a program can contain up to 16,384 (2^{14}) different segments. Segments are of variable size and can range from a single byte to 64K (2^{16}) bytes. So a program's virtual address space can include as much as a full Gb ($2^{30} = 2^{14} \times 2^{16}$) of individually addressable byte locations.

A program's total virtual address space is then subdivided into two distinct halves, the global address space and the local address space, as indicated by the Table Indicator (TI) bit in the virtual address.

The global address space is for system-wide data and procedures, such as operating system software, library routines, runtime language support, and other frequently shared system services. (The operating system is regarded by application programs as a set of service routines that can be accessed by all tasks.) All tasks share global space to prevent unnecessary duplication of system service routines and to promote shared data and interrupt handling. Global address space, defined by addresses with a zero in the TI bit position, is mapped the same for all system tasks.

The other half of the virtual address space, which contains those addresses with the TI bit set, is individually mapped for each system's task. It is called a local address space, because such an address space is local to the task for which it is specified. Generally, code and data segments within a task's local address space are exclusive to that specific task or user. The task isolation that results by dividing the virtual address spaces into local and global regions is shown in Figure 6-2.

Figure 6-2 Address Spaces and Task Isolation



Within each of the two regions (the global address space or a certain local address space) that a program can address, up to 8,192 (2^{13}) different segments can be defined. The segment selector's INDEX field provides for an individual specification of each of these segments. This 13-bit quantity serves as an index into a memory-resident table, or a descriptor table, that keeps a record of the mapping between segment address and the physical locations assigned to each distinct segment. (The sections below discuss these descriptor tables and their purpose in virtual-to-physical address translation.)

In conclusion, a protected mode virtual address is a 32-bit pointer to a specific byte location within a 1-Gb virtual address space. A 16-bit selector component and a 16-bit offset component are included in each such pointer. Alternately, the selector component contains a 13-bit table index, a 1-bit table indicator (local as opposed to global), and a 2-bit RPL field. With the exception of the last field, they all choose a specific segment out of the 16K segments in a task's virtual address space. A full pointer's offset component is an unsigned 16-bit integer that identifies the preferred byte location within the selected segment.

DESCRIPTOR TABLES

A descriptor table is a memory-resident table that is identified by program development tools in a static system or manipulated by operating system software in reprogrammable systems. The contents of the descriptor table control the virtual addresses' interpretation. The 80C286 always references one of these tables whenever it translates a virtual address, thereby decoding a full 32-bit pointer into an equivalent 24-bit physical address.

The global descriptor table (GDT) is one of several descriptor tables usually in memory within a protected mode system. The GDT supplies a full description of the global address space. There may also be one or more local descriptor tables (LDTs), each with a complete description of the local address space of one or more tasks.

For every system task, two descriptor tables (the GDT, which all tasks share, and a specific LDT, exclusive to the task or to a group of similar tasks) supply a full description of that task's virtual address space. Chapter 7, describes the protection mechanism which ensures that a task receives access only to its own virtual address space. Tasks in a simple system configuration can exist completely within the GDT without using local descriptor tables. This simplifies system software because only maintenance of one table (the GDT) is necessary, at the cost of no isolation between tasks. In short, the 80C286 memory management scheme is adaptable enough to allow multiple implementations, and when implementing a system, it does not need to use all possible facilities.

The descriptor tables contain a group of 8-byte entries, from as few as 1 to as many as 8192 in number, called descriptors.

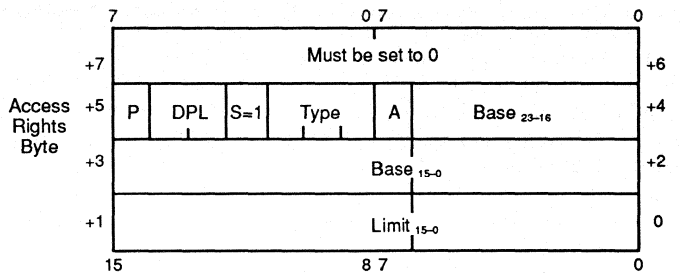
The 80C286 architecture recognizes two main groups of descriptors within a descriptor table. The segment descriptors are the most significant of these, from the consideration of memory management, because they indicate the set of segments that are contained within a given address space. Call gates and task descriptors are among the special-purpose control descriptors in the other group. They impose protection and special system data segments. A segment descriptor's format is illustrated in Figure 6-3.

Note: It supplies information concerning the physical-memory base address, size of a segment, and certain access information. If a certain segment is to be con-

tained within a virtual address space, then a segment descriptor that describes that segment must be contained within the relevant descriptor table. Because of this, there are segment descriptors within the GDT for all of the segments that compose a system's global address space. Likewise, there must be a descriptor within a task's LDT for each segment that is to be contained in that task's local address space.

Each local descriptor table, actually a special system segment, is acknowledged by the 80C286 architecture as such and described by a certain type of segment descriptor as shown in Figure 6-4. Since only a single GDT segment exists, a segment descriptor does not define it. Rather, its base and size information is kept in a dedicated register, GDTR.

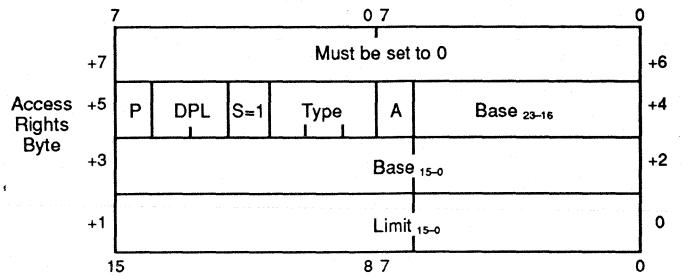
Figure 6-3 Code or Data Segment Descriptor (S = 1)



Access Rights Bytes:
P = Present
DPL = Descriptor Privilege Level
S = Segment Descriptor
Type= Segment Type and Access Information (See Figure 6-7)
A = Accessed

14554A-031

Figure 6-4 Special Purpose Descriptors or System Segment Descriptors (S = 1)



Access Rights Bytes:
P = Present
DPL = Descriptor Privilege Level
S = Segment Descriptor
Type= Segment Type and Access Information (Includes control and system segments)
0 = Invalid Descriptor
1 = Available Task State Segment
2 = LDT Descriptor
3 = Busy Task State Segment
4–7 = Control Descriptor (see Chapter 7)
8 = Invalid Descriptor
9–F = Reserved

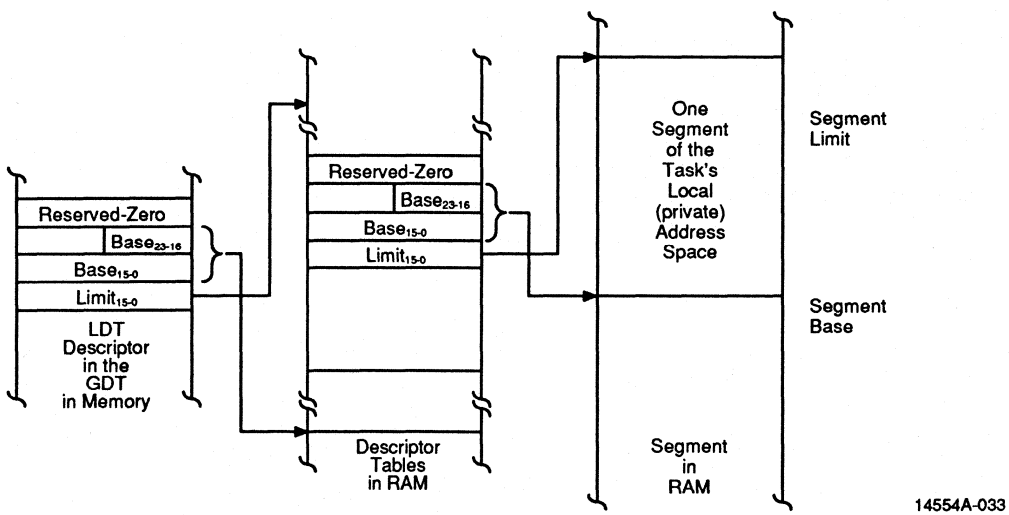
14554A-032

Likewise, LDTR, another dedicated register within the 80C286, records the base and size of the present LDT segment (the LDT associated with the task presently executing). However, the condition of the LDTR register state is volatile. Its contents are automatically modified whenever a task switch from one task to another occurs. Thus, there must be an alternate specification independent of changeable register contents for each LDT in the system. This independent specification is achieved by special system segment descriptors called descriptor table descriptors, or LDT descriptors.

The format of a descriptor table descriptor is shown in Figure 6-4. (Note: It differs from a regular segment descriptor in the contents of certain bits in the access byte.) This particular type of descriptor specifies the physical base address and size of a local descriptor table, which defines the virtual address space and address mapping for a distinct user or task (see Figure 6-5).

Each LDT segment within a system must reside in that system's global address space. All descriptor table descriptors, therefore, must be included among the entries in the system's global descriptor table (the GDT). Only in the GDT may these special descriptors emerge. If reference is made to an LDT descriptor within an LDT, a protection violation will occur. Although they reside in the global address space that is available to all tasks, the descriptor table descriptors are shielded from corruption within the GDT because, as special system segments, they can be accessed only for loading into the LDTR register.

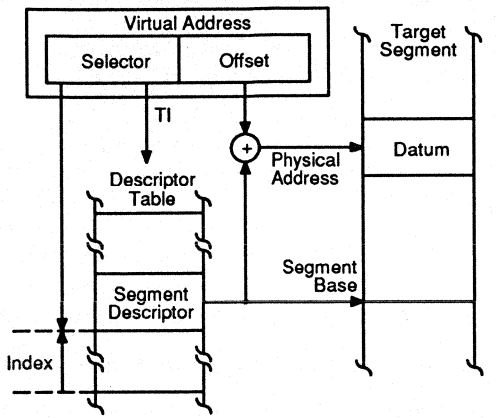
Figure 6-5 **LDT Descriptors**



VIRTUAL-TO-PHYSICAL ADDRESS TRANSLATION

Figure 6-6 illustrates the decoding of a full 32-bit virtual address pointer into an actual 24-bit physical address. When the segment's base address is indicated as a result of the mapping process, the offset value and the result are combined to get the physical address.

Figure 6-6 **Virtual-to-Physical Address Translation**



The mapping itself is executed on the virtual address' selector component. The 16-bit segment selector is mapped to a 24-bit segment base address through a segment descriptor stored in one of the descriptor tables.

The TI bit in the segment selector (see Figure 6-1) specifies which of the descriptor tables (the GDT or the present LDT) is to be selected for memory mapping. Whatever the case, the processor, using the GDTR or LDTR register, can easily determine the memory-resident table's physical base address.

The segment selector's INDEX field identifies a certain descriptor entry within the selected table. The processor multiplies this index value by 8 (a descriptor's length), and then adds the result to the descriptor table's base address to access the applicable segment descriptor in the table.

Lastly, the segment descriptor includes the target segment's physical base address and size (limit) and access information. The processor adds the 24-bit segment base and the identified 16-bit offset to produce the resulting 24-bit physical address.

SEGMENTS AND SEGMENT DESCRIPTORS

The basic units of 80C286 memory management are segments. Segmentation differs from schemes based on fixed-size pages in that it allows for extremely efficient software implementation. Variable-length segments can be fit to the exact specifications of an application. Furthermore, segmentation is consistent with how a programmer naturally treats his virtual address space. Programmers are urged to partition code and data into clearly defined modules and structures, which are controlled as consistent entities. Thus, the potential for virtual memory thrashing is reduced (minimized). In addition, segmentation removes the restrictions on data structures that span a page (a word that extends past page boundaries).

An associated segment descriptor, which may be present in one or more descriptor tables, defines each segment within an 80C286 system. Its presence in a descriptor table represents the inclusion of its associated segment within the virtual address space that table defines. On the other hand, its absence from a descriptor table indicates that the segment is not present in the corresponding address space.

As Figure 6-3 illustrates, an 8-byte segment descriptor encodes the information about a specific segment listed below:

- Size. This 16-bit field, which comprises bytes 0 and 1 of a segment descriptor, identifies an unsigned integer as the size of the segment in bytes (from 1 byte to 64K bytes).

  Unlike segments in the 8086 or the 80C286 in real address mode (never expressly restricted to less than 64K bytes), protected mode segments are always assigned a particular size value. Together with protection features, this designated size promotes the enforcement of a very preferable, natural rule: Inadvertent accesses to locations past a segment's actual boundaries are not allowed.

- Base. This 24-bit field, which comprises bytes 2 through 4 of a segment descriptor, identifies the physical base address of the segment. Thus, it defines the location of the segment itself within the 16-Mb real memory space. The base can be any byte address inside the 16-Mb real memory space.

- Access. Byte 5 of a segment descriptor is comprised of this 8-bit field. This access byte identifies various additional information concerning a segment, especially regarding the 80C286's protection features. Code segments, for instance, differ from data segments, and particular special access restrictions (Execute-Only or Read-Only, for example) may be identified for segments of each type. Access byte values of 00H or 80H always indicate invalid.

The access byte format for code and data segment descriptors is illustrated in Figure 6-7. Chapter 7 provides further information on the following protection related fields inside an access byte: Conforming, Execute-Only, Descriptor Privilege Level, Expand Down, and Write-Permitted. Their use in imposing protection policies is also examined. The Accessed and Present fields are applied in virtual memory implementations.

MEMORY MANAGEMENT REGISTERS

The 80C286's protected virtual address mode features function at high performance because of extensions to the standard 8086 register set. That section of the expanded register structure that relates to memory management is shown in Figure 6-8.

Segment Address Translation Registers

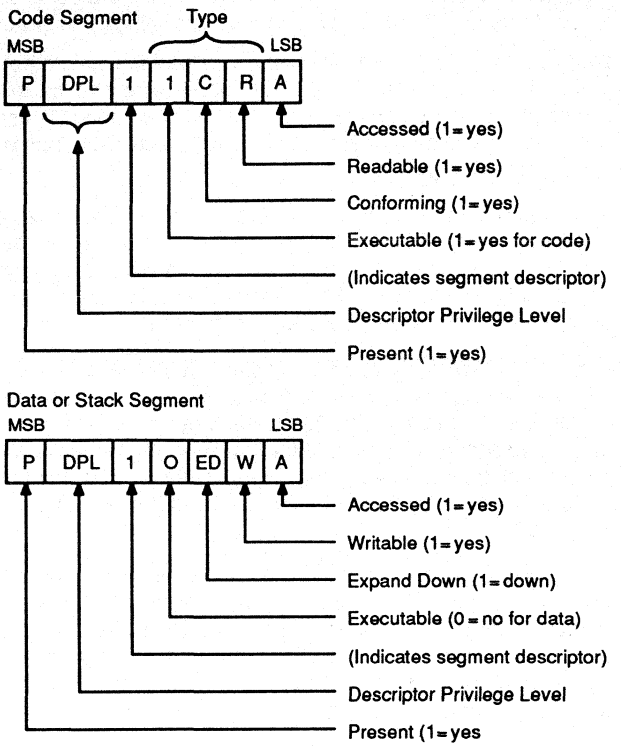
Figure 6-8 presents segment registers CS, DS, ES, and SS. Illustrated as 64-bit registers, each with visible and hidden components, they differ from their normal representation.

Precisely as if they were only the 16-bit segment registers of real address mode, the visible portions of these segment address translation registers are controlled by programs. The program makes the corresponding segment one of its four presently addressable segments by loading a segment selector into one of these segment registers.

The operations that load these registers (that is, those that load the visible part of these registers) are regular program instructions, which can be placed in the categories below:

1. Direct segment-register load instructions. Instructions, such as LDS, LES, MOV, POP, etc., that can clearly reference the SS, DS, or ES segment registers as the destination operand.

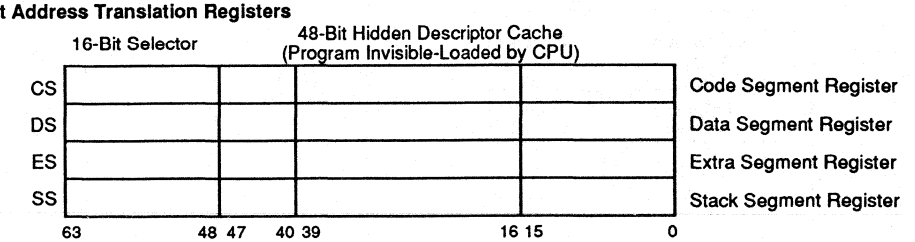
Figure 6-7 Segment Descriptor Access Bytes



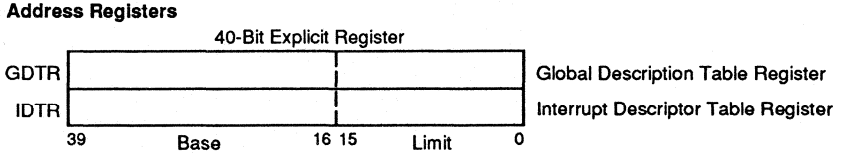
14554A-035

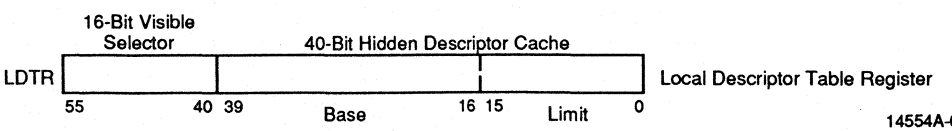
Figure 6-8 Memory Management Registers

Segment Address Translation Registers



System Address Registers





14554A-036

2. Implied segment-register load instructions. Instructions, such as intersegment CALL and JMP, that intuitively reference the CS code segment register. The contents of CS change when these operations are executed.

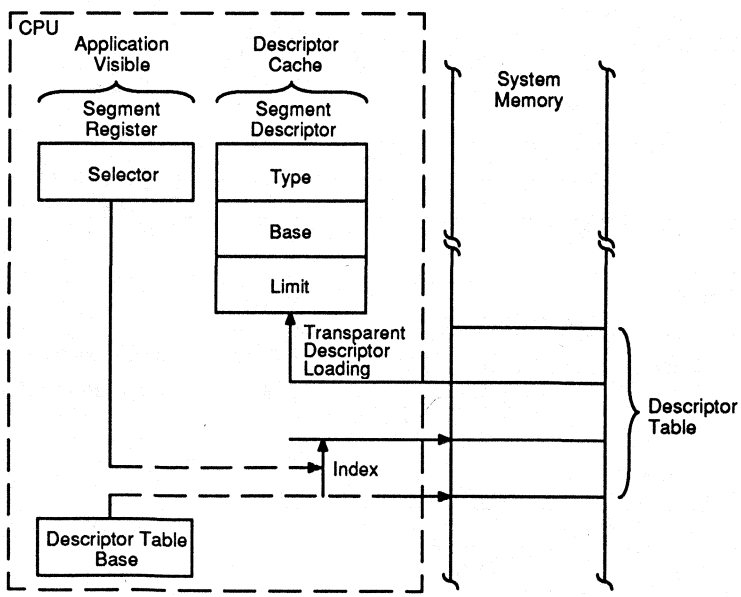
A program, following these directions, can load the visible portion of the segment register with a 16-bit selector (the high-order word of a virtual address pointer). Whenever this process is complete, the processor automatically uses the selector in order to reference the relevant descriptor. Then it loads the 48-bit hidden descriptor cache for that segment register.

The correspondence between selectors and descriptors was discussed previously. Recall that the selector's TI bit shows either one of the two descriptor tables (the LDT or the GDT). The selector's 13-bit INDEX field selects a specific entry within the implied table. The relative displacement of the selected table entry (a descriptor) is represented by this index scaled by a factor of 8.

Therefore, as long as a specific selector value is valid (it is pointing to a valid segment descriptor inside the bounds of the descriptor table), it can be easily associated with an 8-byte descriptor. The 80C286 loads 6 bytes of the associated descriptor into the register's hidden portion automatically whenever a selector value is loaded into a segment register's visible portion. Thus, these 6 bytes include the chosen segment's size, base, and access type. This transparent process of descriptor loading is shown in Figure 6-9.

The segment registers' hidden descriptor fields operate as the 80C286's memory management cache. The memory cache stores all information needed to address the present working set of segments (the currently addressable segments' base address, size, and access rights). The 80C286 cache is entirely deterministic, however, in comparison to the probabilistic caches of other architectures. The caching of descriptors is solely manipulated by the program.

Figure 6-9 **Descriptor Loading**



The translation of a full 32-bit virtual address, or long pointer, is not a requirement of most memory references. Operands located within one of the presently addressable segments, as the four segment registers indicate, can be referenced most effectively with a short pointer, or 16-bit offset.

Actually, the majority of 80C286 instructions reference memory locations this exact way, identifying only a 16-bit offset with regard to one of the presently addressable segments. The choice of segments CS, DS, ESS, or SS is either implied within the actual instruction, or clearly indicated by a segment-override prefix.

In effect, virtual-to-physical address translation is actually executed in two steps in most cases. A program first loads a new value into a segment register and the processor instantly executes a mapping operation. The selected segment's physical base address and certain additional information are automatically loaded into the register's hidden portion. Thus, the internal cache registers, or virtual address translation hardware, are dynamically shared among the 16K separate segments that may be addressed within the user's virtual address space. Software overhead, system or application, is not necessary to perform this operation.

Later, as the program uses a short pointer to reference a location within a segment, the processor produces a 24-bit physical address just by adding the indicated offset value to the segment base address cached before. The 80C286 supplies an effective on-chip mechanism for address translation (with little overhead for references to memory-based tables or required external address-translation devices) by promoting the use of short pointers in this way, instead of requiring a full 32-bit virtual address for each memory reference.

System Address Registers

The Global Descriptor Table Register (GDTR), a dedicated 40-bit (5 byte) register, records the base and size of a system's global descriptor table (GDT). Two of these bytes, therefore, specify the GDT's size. Three bytes specify its base address.

The GDTR's contents are called a hidden descriptor in Figure 6-8. In this instance, the term descriptor stresses the analogy with the segment descriptors normally found in descriptor tables. Much as these descriptors define the base and size, or limit, of normal segments, the GDTR register defines these same parameters for that segment of memory that serves as the system GDT. The limit does not allow accesses to descriptors in the GDT to access past the end of the GDT. Thus, the limit supplies address space isolation at the system and task levels.

The register contents are hidden merely because they are not accessible by ordinary instructions. Rather, the dedicated protected instructions LGDT and SGDT are used only for loading and storing, respectively, the GDTR's contents at protected mode initialization. Further modification of the GDT base and size values is not encouraged, but is a system option at the software's most privileged level.

The Local Descriptor Table Register (LDTR), a dedicated 40-bit register, includes at any time the base and size of the local descriptor table (LDT) that corresponds with the presently executing task. The LDTR register boasts a visible and a hidden component, unlike GDTR. Although the visible component is accessible, the hidden component is still not accessible, even to dedicated instructions.

A 16-bit selector field constitutes the visible component of the LDTR. The format of these 16 bits precisely corresponds to a segment selector's format in a virtual address pointer. Therefore, it includes a 13-bit INDEX field, a 1-bit TI field, and a 2-bit RPL

field. Implying a reference to the GDT, or global address space, the TI bit must be zero. The INDEX field, as a result, supplies an index to a specific entry within the GDT. In turn, this entry must be an LDT descriptor, or descriptor table descriptor, as specified in the section above. Thus, the visible selector field of the LDTR uniquely defines a specific LDT in the system by choosing an LDT descriptor.

The dedicated, protected instructions LLDT and SLDT are used only to load and store, respectively, the LDTR register's visible selector component. Whenever a new value is loaded into the LDTR's visible selector portion, an LDT descriptor will have been distinctly selected, as long as the selector value is valid. In this instance, the 80C286 automatically loads the LDTR's hidden descriptor portion with five bytes from the selected LDT descriptor. Size and base information about a certain LDT, as maintained in a memory-resident global descriptor table entry, is therefore cached in the LDTR register.

New values may be loaded into the LDTR's visible and, consequently, hidden portions in one of two ways. During system initialization, the LLDT instruction is used solely to set an initial value for the LDTR register. By this process, a local address space is supplied for the first task in a multitasking environment. After system startup, formal changes are not necessary, since operations that instantly cause a task switch appropriately control the LDTR.

Thus, the LDTR register constantly records the physical base address (and size) of the current task's LDT. The descriptor table needed for mapping the present local address space is consequently accessible to the processor immediately. Furthermore, because GDTR always maintains the GDT's base address, the table mapping the global address space is also accessible. GDTR and LDTR, the two system address registers, serve as a special processor cache, providing current information about the two descriptor tables needed, at any particular time, to address the complete present virtual address space.

CHAPTER 7

PROTECTION

INTRODUCTION

In the majority of microprocessor based products, the product's software, often responsible for a product's success, determines its availability, quality, and reliability. Protection, a tool used to decrease software development time, enhances software quality and reliability.

Program testing is an important part of software development. With protection, a system can spot software errors quicker and more accurately than a system that does not have protection. Removing errors with protection decreases a product's development time.

Testing software is not an easy task. Many errors result only under complicated circumstances, which cannot be readily anticipated. As a result, products are shipped with undetected errors. Products seem unreliable when such errors happen. If the software error causes errors in other bug-free programs, its impact is multiplied. The total system dependability, as a result, drops to that of the least reliable program operating at any given moment.

Protection enhances an entire system's reliability by preventing one program's software errors from affecting other programs. With protection, the system can keep running even when a user program tries an operation that is incorrect or not allowed.

Hardware protection executes run-time checks that parallel the program's performance. Hardware protection has traditionally resulted, however, in a slower and more expensive design, when compared to a system that does not have protection. The 80C286 supplies hardware-enforced protection, however, minus the performance or cost penalties usually associated with protection.

The protected mode 80C286 imposes vast protection by integrating these functions on-chip. The 80C286 protection, more comprehensive and adaptable than comparable solutions, can locate and isolate a great number of program errors and then prevent the introduction of such errors to other tasks or programs. The total system's protection locates and isolates errors during development and installed usage. The rest of this chapter explains the protection model used in the 80C286.

Types of Protection

Three basic aspects characterize protection in the 80C286:

1. System software is isolated from user applications.
2. Users are isolated from each other (inter-task protection).
3. Data-type checking.

The 80C286 furnishes a four-level, ringed-type, increasingly-privileged protection mechanism to separate applications software from other layers of system software, a major improvement and extension of the easier two-level user/supervisor mechanism

of many systems. Supervisor level software modules are protected from application level modules and from software in less privileged supervisor levels.

Restricting a software module's addressability allows an operating system to manage system resources and priorities, which is particularly important in an environment of multiple simultaneous users. This total control of system resources for efficient, dependable operation is necessary for multi-user, multi-tasking, and distributed processing systems.

Another aspect of protection is separating users from each other. Without this aspect of protection, an error in one user program could affect the performance of another error-free user program. Such subtle interactions are difficult to find and fix, and thus, the reliability of applications programs is largely improved by such separation of users.

Within a system or application level program, the 80C286 ensures that all code and data segments are utilized properly (data cannot be executed, programs cannot be altered, offset must be inside identified limits, etc.). In order to provide complete run-time error checking, such checks are executed on each memory access.

Protection Implementation

The 80C286's protection hardware generates constraints on memory and instruction usage. The number of possible interactions between instructions, memory, and I/O devices is almost endless. Out of this very large field, the protection mechanism restricts interactions to a confined, comprehensive subset. The list of correct operations falls inside this subset. Any operation that falls outside this subset is prohibited by the protection mechanism and is labeled as a protection violation.

To understand 80C286 protection, start with its basic parts: segments and tasks. 80C286 segments are the smallest region of memory with distinct protection attributes. Modular programming automatically generates distinct regions of memory (segments). The contents of these segments are regarded as a whole. Segments display the natural construction of a program (code for module A, data for module A, stack for the task, etc.). The 80C286 treats all parts of the segment equally. Logically, each region of memory should be in an individual segment.

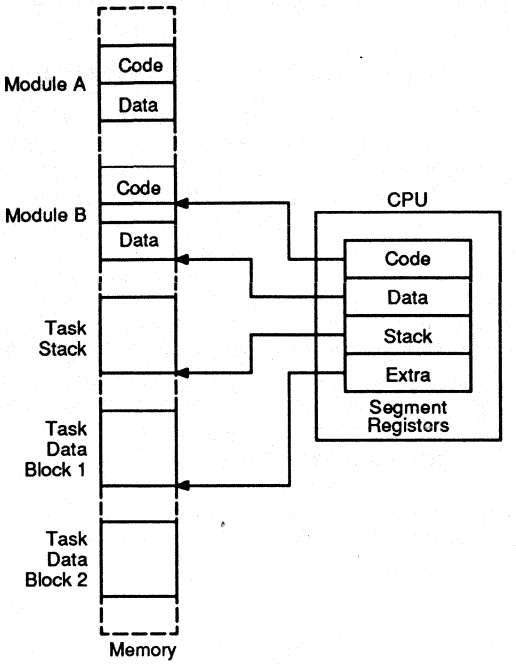
The 80C286's memory segmentation model, shown in Figure 7-1, was designed to best execute code for software with independent modules. Modular programs are not as difficult to construct and support. Modular software systems, versus monolithic software systems, have improved capabilities, and are generally easier to formulate and test for appropriate operation.

A memory-resident descriptor identifies every segment in the system. The protection hardware prohibits accesses beyond the data areas and tries to alter instructions, etc., as specified by the descriptors. On the 80C286, segmentation permits protection hardware to be integrated into the CPU for complete data access control with no performance impact.

The 80C286's segmented memory architecture furnishes distinct capabilities for managing the transfer of control between programs.

Programs receive direct but restricted access to other procedures and modules. This ability is the center of isolating application and system programs. Because this access is supplied and controlled directly by the 80C286 hardware, no performance penalty

Figure 7-1 **Addressing Segments of a Module within a Task**



14554A-039

results. A system designer can use the 80C286 access control to create high-performance modular systems with great confidence in the system's integrity.

Access control between programs and the operating system is imposed through address space separation and a privilege mechanism. The address space control isolates applications programs from each other. The privilege mechanism, which separates system software from applications software, gives different capabilities to programs to access code, data, and I/O resources, which depend on the associated protection level. Trusted software that manages the entire system is generally set at the greatest privileged level. These control mechanisms, which have no effect on ordinary application software, become part of the picture only when there is a transfer of control between tasks, or if the operating system routines must be activated.

The protection features of several privilege levels also ensure dependable I/O control. However, if a system designer enabled only one specific level to do I/O, it would exceedingly restrict further extensions or application development. Rather, the 80C286 lets each task be given a separate minimum level where I/O is allowed.

There is a significant distinction between tasks and programs. Programs (instructions within code segments) are static and include a fixed set of code and data segments, each with a corresponding privilege level. The privilege given to a program establishes what the program may do when a task executes it. A program receives privilege when the system is built or the program loaded.

Tasks are dynamic. That is, they perform one or more programs. Task privilege is modified with time, based on the privilege level of the program being executed. Every task has a distinct group of attributes that define it: address space, register values,

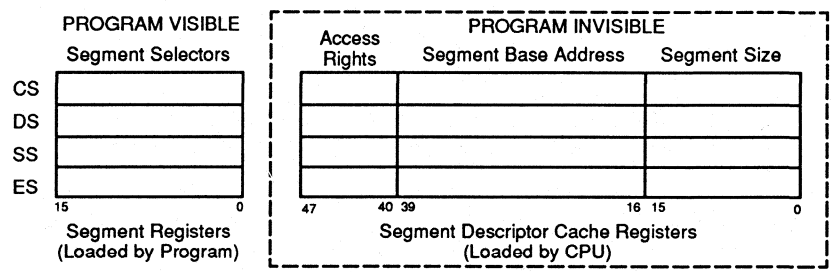
stack, data, etc. A task can perform a program if that program is found in the task's address space. Protection control rules indicate when a task may execute a program. Once the program is executed, these rules decide what its capabilities are.

MEMORY MANAGEMENT AND PROTECTION

The 80C286's protection hardware corresponds with the memory management hardware. Because protection attributes are given to segments, they are maintained along with the memory management information in the segment descriptor. When the segment is generated, the protection information is provided. Besides privilege levels, the descriptor also identifies the segment type (code segment, data segment, etc.). Descriptors may be produced in one of two ways: by program development tools or by a loader in a dynamically loaded reprogrammable environment.

The protection control information includes a segment type, its privilege level, and its size, all of which are fields in the access byte of the segment descriptor (see Figure 7-2). For quick access during execution, this information is saved on-chip in the programmer invisible section of the segment register, and altered whenever a segment register is loaded. The protection data is used twice, once upon loading a segment register, and again upon each reference to the selected segment.

Figure 7-2 Descriptor Cache Registers



09729B-14

While loading a segment register, the hardware executes multiple checks, which enforce the protection rules before any memory reference is produced. The hardware confirms that the chosen segment is valid (that is, it is specified by a descriptor, is in memory, and can be accessed from the privilege level of the executing program), and that the type is the same as the target segment register. A read-only segment descriptor, for example, cannot be loaded into SS since the stack must constantly be writable.

Every reference into the segment indicated by a segment register is checked by the hardware to confirm that it is within the specified limits of the segment and is of the appropriate type. A code segment or read-only data segment, for instance, is not writable. All these checks are made prior to the start of the memory cycle, and any violation will not let that cycle start and, thus, an exception will result. Since the checks are executed simultaneously with address formation, no performance penalty occurs.

The system designer can assure a program will not modify its code or overwrite data that belongs to another task simply by manipulating the access rights and privilege

attributes of segments. Such assurances are crucial to preserving system integrity when error-prone programs exist.

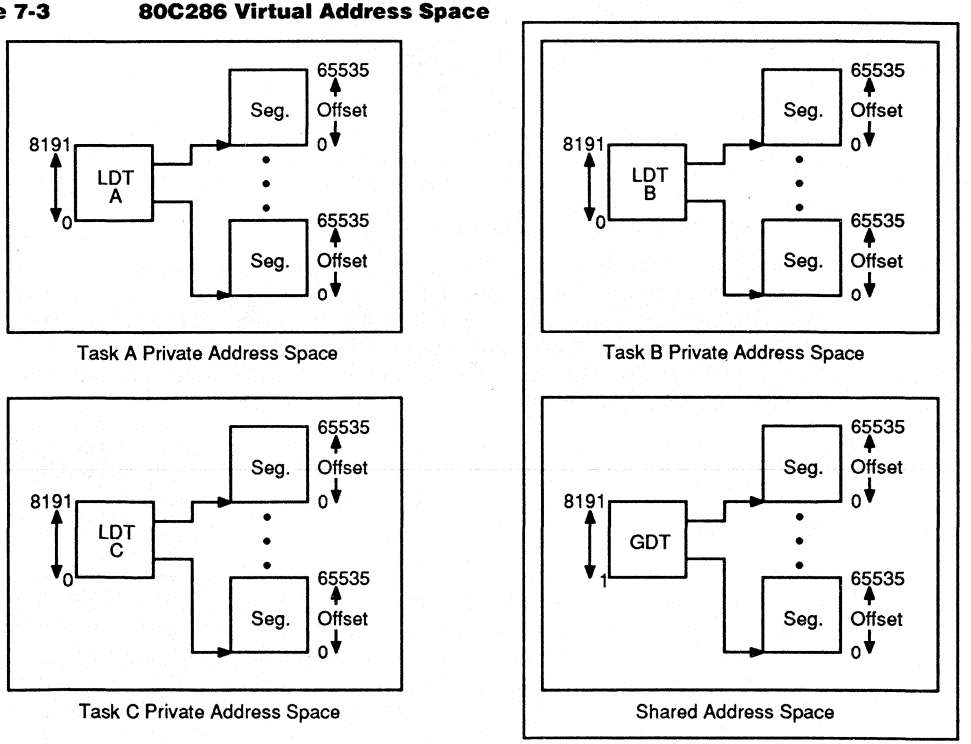
Separation of Address Spaces

Every task can address up to a Gb ($2^{14}-2$ segments of as many as 65,536 bytes each) of virtual memory determined by the task's LDT (Local Descriptor Table) and the system GDT. Up to one-half Gb (2^{13} segments of as many as 65,536 bytes each) of the task's address space is determined by the LDT and denotes the task's private address space. The rest of the virtual address space, determined by the GDT, is standard to all system tasks.

Every descriptor table defined by descriptors in the GDT (Global Descriptor Table) is actually a special kind of segment that the 80C286 architecture recognizes. The CPU has a set of base and limit registers pointing to the GDT and the LDT of the presently executing task. A task switch operation loads the local descriptor table register.

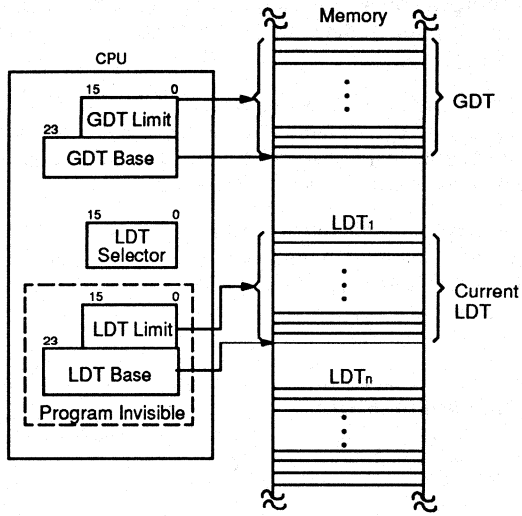
An active task may load only those selectors referencing segments, as defined by descriptors in either the GDT or its private LDT. Since a task is not able to reference descriptors in other LDTs, and no descriptors in its LDT concern data or code that belong to other tasks, access to another task's private code and data (see Figure 7-3) is not possible.

Figure 7-3 80C286 Virtual Address Space



Because the GDT includes information accessible by all users (library routines, common data, operating system services, etc.), the 80C286 utilizes privilege levels and special descriptor types to manage access. While privilege levels shield more trusted data and code (in GDT and LDT) from less trusted access WITHIN a task, private virtual address spaces determined by distinct LDTs supply protection BETWEEN tasks (see Figure 7-4).

Figure 7-4 **Local and Global Descriptor Table Definitions**



09829B-015

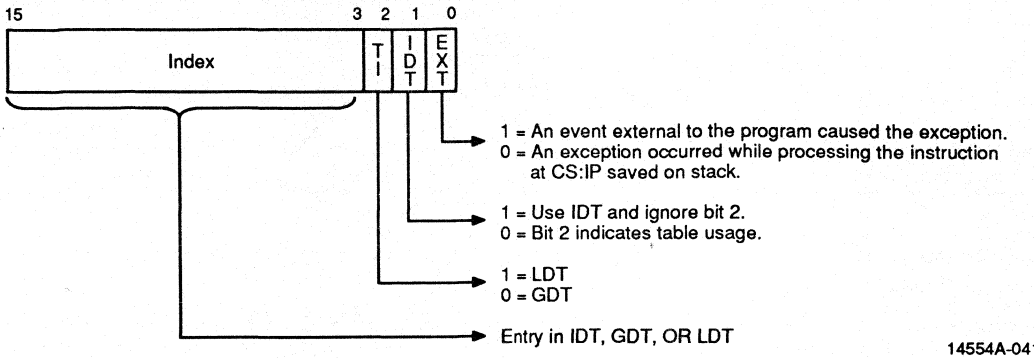
LDT and GDT Access Checks

To allow for address space separation of tasks, the protection hardware uses a limit for all descriptor tables. Each task's LDT can vary in size as specified by its descriptor in the GDT. The GDT may also include less than 8191 descriptors as specified by the limit value of the GDT. The descriptor table limit specifies the last valid byte of that table's last descriptor. The limit value is $(N \times 8) - 1$ for N descriptors because each descriptor is eight bytes in length.

If a program tries to load a segment register, local descriptor table register (LDTR), or task register (TR) with a selector that concerns a descriptor outside the associated limit, an exception with an error code defining the invalid selector used will occur (see Figure 7-5).

Every descriptor entry in the GDT or LDT does not need a valid descriptor. Holes, or empty descriptors, can exist in the LDT and GDT. Empty descriptors allow segments or other system objects to be dynamically appropriated and deleted without altering the size of the GDT or LDT. A descriptor is considered empty if it has an access byte that is zero. If an attempt is made to load a segment register with a selector that concerns an empty descriptor, an exception with an error code defining the invalid selection will occur.

Figure 7-5 **Error Code Format (on the stack)**



Type Validation

After determining that a selector reference is inside the bounds of a descriptor table and concerns a non-empty descriptor, the segment type, determined by the descriptor, is checked against the destination register. Every segment register with its functions previously defined must refer to certain types of segments. If an attempt is made to load a segment register that violates protection rules, an exception will occur.

The null selector, a special type of segment selector, has an index field of all zeros and a table indicator of 0, and appears to refer to GDT descriptor entry #0 (see GDT in Figure 7-3). This selector value, which may be used as a place holder in the DS or ES segment registers, may be loaded into them without an exception resulting. An exception will occur, however, and prevent any memory cycle from happening, if any attempt is made to use the null segment registers to reference memory.

PRIVILEGE LEVELS AND PROTECTION

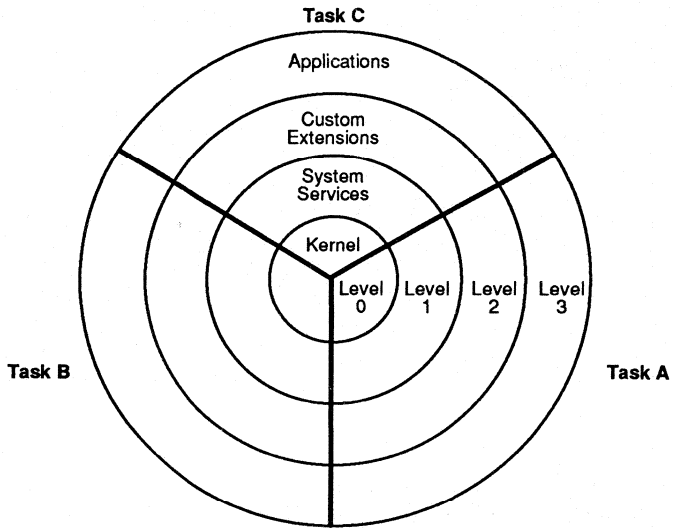
Every task has its own distinct virtual address space, as determined by its LDT. The GDT specifies a standard address space that all tasks share. As a result, the system software can directly access task data and treat all pointers the same.

Protection is necessary to prohibit programs from inappropriately using code or data belonging to the operating system. The four privilege levels of the 80C286 supply the necessary separation between the multiple system layers. The 80C286 privilege levels are numbered from 0 to 3; 0 is the most trusted level, 3 the least.

Privilege level, a protection attribute all segments receive, decides which procedures may access the segment. Privilege checks, like access rights and limit checks, are automatically executed by the hardware, and thus both data and code segments are protected.

Hierarchical privilege exists on the 80C286 in such a way that programs at other privilege levels do not have access to operating system code and data segments set at the most privileged level (0). Programs at privilege level 0, however, have access to data at all other levels. Data at the same or less trusted (numerically greater) privilege levels is accessible only to programs at privilege levels 1–3. The privilege level protection of code or data within tasks is shown in Figure 7-6.

Figure 7-6 **Code and Data Segments Assigned to a Privilege Level**



Programs can access data at the same or outer level in Figure 7-6, but data is not accessible at inner levels. Code and data segments set at level 1 are not accessible by programs performing at levels 2 or 3. While servicing level 1, programs at privilege level 0 can access data at level 1. The 80C286 supplies mechanisms for inter-level transfer of control when necessary.

The 80C286's four privilege levels expand the common two-level user/supervisor privilege mechanism. Application programs in the outer level, like user mode, are not allowed direct access to data that belongs to more privileged system services (supervisor mode). The 80C286 includes two other privilege levels to furnish protection for various layers of system software (system services, I/O drivers, etc.).

Example of Using Four Privilege Levels

Two additional privilege levels promote development of more dependable and adaptable system software by splitting the system into small, independent units. An example of the utilization of various protection levels is illustrated in Figure 7-6, where the most privileged level is named the kernel. Standard, application-independent, CPU-oriented services to all tasks would be supplied by this software. Memory management, task isolation, multi-tasking, inter-task communication, and I/O resource control are among such services. The kernel, only concerned with simple functions and not subject to software at other privilege levels, can be kept small, sound, and comprehensive.

Designated system services comprise privilege level 1. This software furnishes high-level functions, such as file access scheduling, character I/O, data communications, and resource allocation policy, which are expected as standard in all systems. Such software, still kept separate from applications programs, depends on the kernel's services, yet cannot affect level 0's integrity.

The custom operating system extensions level comprises privilege level 2, which lets standard system software be customized. Although such customizing can be kept separate from errors in applications programs, it cannot alter the system software's basic integrity. The data base manager, logical file access services, etc. are examples of customized software.

This is only one instance of protection mechanism usage. Levels 1 and 2 can be used in several different ways, and the system designer determines their usage (or non-usage).

Even though programs at all privilege levels are separated from programs at outer layers, they have no effect on programs in inner layers. Programs written for each privilege level can be smaller and easier to conceive and maintain than a monolithic system, in which all system software can affect all other system software.

Privilege Usage

Privilege extends to tasks and three groups of descriptors, listed below.

1. Main memory segments
2. Gates (control descriptors for state or task transitions)
3. Task state segments

Task privilege, a dynamic value, comes from the code segment that is being presently executed. A task privilege may be modified only when a control transfers to a different code.

Descriptor privilege, which includes code segment privilege, is designated when the descriptor, and any corresponding segment, is generated. The system designer directly allocates privilege when the system is built with the system builder or indirectly through a loader.

Each task performs at only one privilege level at any given instance: mainly, that of the executing code segment. (The conforming segments allow for some flexibility in this regard.) The task, however, may include segments at one, two, three, or four levels, all of which are to be used at proper times, as shown in Figure 7-6. Then, the task's privilege level transforms under the cautiously enforced rules for transfer of control from one code segment to another.

The descriptor privilege attribute is kept in a descriptor's access byte and is known as the Descriptor Privilege Level (DPL). Task privilege is referred to as the Current Privilege Level (CPL) and is identified by the CS register's least important two bits.

Before examining the detailed descriptions of later sections, a few basic rules of privilege can be set. Data access is limited to data segments with privilege levels equally or less privileged (numerically greater) than the current privilege level (CPL). Direct code access, through call or jump, is confined to code segments of the same privilege. A gate is necessary for access to code at higher privilege levels.

SEGMENT DESCRIPTOR

Although access control information's format, described below, is alike for both data and code segment descriptors, the rules for accessing data segments vary from those for switching control to code segments. Data segments are accessible from many privilege levels, that is, from programs at the same level or from deeper in the operating system. The primary limitation is that less privileged code cannot access them.

Code segments, however, are meant to be performed at only one privilege level. Control transfers past privilege boundaries are firmly restricted. Thus, the use of gates is necessary. Likewise, control transfers within a privilege level may also use gates, but they are not necessary.

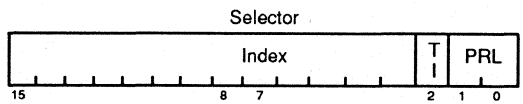
Protection checks are automatically activated at several points in choosing and using new segments. The process of addressing memory starts when the presently executing program tries to load a selector into one of the segment registers. The selector has the form indicated in Figure 7-7.

The processor accesses the corresponding descriptor to execute the required loading and privilege checks whenever a new selector is loaded into a segment register.

The protection mechanism confirms that the selector points to a valid descriptor type for the segment register. Following confirmation of the descriptor type, the CPU contrasts the task's privilege level (CPL) to the descriptor's privilege level (DPL) prior to loading the descriptor's information into the cache.

Table 7-1 illustrates the basic format of the eight bits in the segment descriptor's access rights byte.

Figure 7-7 **Selector Fields**



Bits	Name	Function
1–0	Requested Privilege Level (RPL)	Indicates Selector Privilege Level Desired
2	Table Indicator (TI)	TI = 0 Use Global Descriptor Table (GDT) TI = 1 Use Local Descriptor Table (LDT)
15–3	Index	Select Descriptor Entry in Table

09729B-013

Table 7-1 **Segment Access Rights Byte Format**

Bit	Name	Description
7	Present	1 means Present and addressable in real memory; 0 means not present.
6, 5	DPL	2-bit Descriptor Privilege Level, 0 to 3.
4	Segment	1 means Segment descriptor; 0 means control descriptor.
For Segment = 1, the remaining bits have the following meanings:		
3	Executable	1 means code; 0 means data.
2	C or ED	If code, Conforming: 1 means yes, 0 no. If data, Expand Down: 1 means yes, 0 no (normal case).
1	R or W	If code, Readable: 1 means readable, 0 not. If data, Writable: 1 means writable, 0 not.
0	Accessed	1 if segment descriptor has been Accessed, 0 if not.

Note: When the Segment bit (bit 4) is 0, the descriptor is for a gate, a task state segment, or Local Descriptor table, and the meanings of bits 0–3 change.

Figure 7-8 shows the access rights byte for a data and code segment in real memory, but not yet accessed at the same privilege level. The accessed bit in the descriptor table is set to 1 when a segment descriptor is loaded into a segment register. This bit aids in designating the segment's usage profile.

Data Accesses

Data is accessible in data segments or readable code segments. Whenever DS or ES is loaded with a new selector (by an LDS, LES, or MOV to ES, SS, or DS register), the bits in the access byte are checked to confirm valid descriptor type and access (see Table 7-2). If a test fails, an error code is pushed onto the stack specifying the selector concerned (Figure 7-5 shows the error code format).

When the segment register is loaded, a privilege check is performed. Typically, a data segment's DPL must be numerically larger than or the same as the CPL, and the DPL of a descriptor loaded into the SS must match the CPL. An exception to privilege checking rules is conforming code segments.

Figure 7-8 **Access Byte Examples**

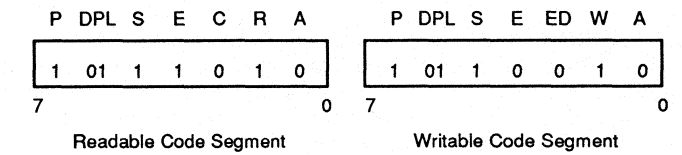


Table 7-2 **Allowed Segment Types in Segment Registers**

	Allowed Segment Types			
Segment Register	Read Only Data Segment	Read-Write Data Segment	Execute Only Code Segment	Execute-Read Code Segment
DS	Yes	Yes	No	Yes
ES	Yes	Yes	No	Yes
SS	No	Yes	No	No
CS	No	No	Yes	Yes

Once the segment descriptor and selector are loaded, the offset of later accesses within the segment are verified against the boundary provided by the segment descriptor. Ignoring the segment size restriction results in a general protection exception with an error code of 0.

A regular data segment is addressed with offset values that range from 0 to the segment's size. The designated range of offsets is 0000H to the limit when the ED bit of the access rights byte in the segment descriptor equals 0. The data segment has 65,536 bytes if limit equals 0FFFFH.

Since stacks typically occupy offset ranges (lower limit to 0FFFFH) that differ from data segments, a segment descriptor's limit field can be understood in two ways. The Expand Down (ED) bit in the access byte lets offsets for stack segments surpass the limit field. When ED equals 1, the permitted range of offsets within the segment is limit +1 to 0FFFFH. Set ED to 1 and the limit to 0FFFFH to permit a complete stack segment. For use in SS, the ED bit of a data segment need not be set (an exception will not occur). An expand down segment, in which ED equals 1, can also be loaded into ES or DS.

Limit and access checks are executed before beginning any memory reference. For stack push instructions (PUSH, PUSHA, ENTER, CALL, INT), a conceivable limit violation is specified prior to any internal register updates. Thus, after a stack size violation, these instructions are entirely restartable.

Code Segment Access

Code segments are accessed through CS to be executed. Segments labeled execute-only can ONLY be executed. Execute-only segments cannot be accessed through DS or ES, nor read through CS with a CS override prefix. If a segment can be executed (i.e., if bit 3 equals 1 in the access byte), it must also be readable for access through DS or ES to be possible. Any code segment that also includes data, therefore, must be readable.

An execute-only segment maintains the code's privacy against any effort to read it. A general protection fault with an error code of 0 results if such an attempt is made. It is not possible to load a code segment into SS, and such a segment is never writable. A general protection fault with an error code of 0 will occur if any attempted write is made.

The limit field of a code segment descriptor specifies the segment's final byte. A general protection fault will occur if any offset higher than the limit value exists. The 80C286's prefetcher cannot possibly cause a code segment limit violation with an error code of 0. To cause an exception, the program must actually try to perform an instruction past the end of the code segment.

The privilege level requirements are identical to those stated for data segments as long as a readable non-conforming code segment is to be loaded into DS or ES.

When performed, code segments are exposed to different privilege checks. The present privilege level must equal the new code segment's descriptor privilege level to meet the regular privilege requirement for a jump or call to another code segment. Jumps and calls inside the present code segment automatically comply with this rule.

Return instructions may grant control to code segments at the equal or less (numerically greater) privilege level.

As discussed earlier, the conforming code segment that lets the DPL of the desired code segment be numerically less than (of higher privilege than) the CPL is an exception to this.

Data Access Restriction by Privilege Level

Privilege confirmation when accessing either data segments (loading segment selectors into DS, ES, or SS) or code segments that are readable is discussed in this section. Privilege confirmation when loading CS to transfer control across privilege levels is the topic of the next section.

When determining accessibility to a segment for reading and writing, three standard types of privilege level indicators are used: Current Privilege Level (CPL), Descriptor Privilege Level (DPL), and Requested Privilege Level (RPL). The CPL, stored as bits 0 and 1 of the CS and SS registers, is merely the privilege level of the executing code segment (except if the present code segment is complying). Unrelated to CPL are bits 0 and 1 of DS and ES.

DPL, stored in bits 5 and 6 of a descriptor's access byte, is the segment's privilege level. For data access to data segments and non-conforming code segments to be granted, CPL must be numerically less than or the same as DPL (the task must be of the same or higher privilege). If this rule is broken during segment load instruction, a general protection exception with an error code specifying the selector results.

While the enforcement of DPL protection rules supplies the mechanism for the isolation of code and data at different privilege levels, it is possible that the illegal alteration of data with a higher privilege level might occur if an erroneous pointer was passed onto a more trusted program. Enforcing effective privilege level protection rules and valid usage of the RPL value prevents this possibility.

The Requested Privilege Level (RPL), the least significant two bits in the selector value loaded into any segment register, is used for pointer validation. RPL is meant to specify the privilege level of that selector's originator. A selector may be passed down through multiple procedures at varying levels. The RPL depicts the privilege level of the selector's original supplier, not its intermediate supplier. Thus, the RPL, indicating higher or identical privilege of the supplier, must be numerically less than or identical to the DPL of the chosen descriptor. Access is denied and a general protection violation results if this is not the case.

Any system that is concerned with preventing program errors from shattering system integrity requires pointer validity testing. The 80C286 furnishes hardware support for such testing. The privilege level of the originator of the pointer to the hardware is identified in the RPL field. If the CPL is numerically less than or identical to the DPL, access will still be denied if the pointer's originator did not have access to the chosen segment. RPL can decrease a task's effective privilege when using a certain selector. RPL never permits access to more privileged segments. Rather, CPL must constantly be numerically less than or identical to DPL.

A fourth term, the Effective Privilege Level (EPL), is occasionally used. The EPL is the numeric maximum of the CPL and the RPL (that is, the one of lesser privilege). Access to a protected entity is allowed only when the EPL is numerically less than or identical to that entity's DPL. In other words, both CPL and RPL must be numerically less than or identical to DPL to allow access.

Pointer Privilege Stamping via ARPL

The 80C286 supplies the ARPL instruction to fill the RPL field of a selector with the least privilege (maximum numeric value) of the selector's present RPL and the caller's CPL (provided in an instruction-specified register). A straight insertion of the caller's CPL would mark the pointer with the caller's privilege level, but not necessarily the final originator of the selector. (Level 3 provides a selector to a level 2 routine that calls a level 0 routine with the identical selector.)

A program with an example of such a situation is shown in Figure 7-9. The program at privilege level 3 calls a routine at level 2 through a gate, and the routine at level 2 utilizes the ARPL instruction to affirm that the selector's RPL is 3. When the level 2 routine calls a routine at level 0 and passes the selector, the RPL field is left unchanged by the ARPL instruction at level 0.

Marking a pointer with the originator's privilege disposes of the complicated and time-consuming software usually associated with pointer confirmation in architectures that are less comprehensive. While loading the selector, the 80C286 hardware executes the pointer test automatically.

Privilege errors are confined when the selector is loaded because pointers are generally passed to other routines, and it may not be possible to specify a pointer's originator. To confirm a pointer's access capabilities, it should be tested whenever the pointer is received from an untrusted source. The VERR (Verify Read), VERW (Verify Write), and LAR (Load Access Rights) instructions are furnished for this intent.

Although pointer confirmation is entirely supported in the 80C286, its use is optional for the system designer. To reconcile systems in which it is not necessary, RPL can be disregarded by setting selector RPLs to zero (not including stack segment selectors) and not modifying them with the ARPL instruction.

Figure 7-9 Pointer Privilege Stamping

```
Level 3         PUSH    SELECTOR        ; RPL value doesn't matter at level 3
                CALL    LEVEL_2
-------------------------------------------------------------------------------
            Level_2:
                ENTER   4,0
                MOV     AX, [BP]+4      ; GET CS of return address, RPL=3
                ARPL    [BP]+6, AX      ; Put 3 in RPL field
Level 2           .
                  .
                  .
                PUSH    WORD PTR [BP]+6; Pass selector
                CALL    Level_0
-------------------------------------------------------------------------------
            Level_0:
                ENTER   6,0
Level 0         MOV     AX, [BP]+4      ; Get CS of return address, RPL=2
                ARPL    [BP]+6, AX      ; Leaves RPL unchanged
```

CONTROL TRANSFERS

Three types of control transfers that are possible within a task are listed below:

1. Within a segment (a *short* jump, call, or return), privilege level remaining unchanged.
2. Between segments at the identical privilege level (a *long* jump, call, or return).
3. Between segments at various privilege levels (a *long* call, or return). A JUMP to a different privilege level is not permitted.

Beyond those described in the Segment Descriptor section of this chapter, no special controls (with respect to privilege protection) are required for the first two types of control transfers.

To preserve system integrity, special consideration must be given to inter-level transfers. The protection hardware must examine the following:

- The task is presently permitted to access the destination address.
- The correct entry address is utilized.

To accomplish control transfers, a special descriptor type, known as a gate, is supplied to arbitrate the change in privilege level. Control transfer instructions call the

gate instead of transferring directly to a code segment. A control transfer to a gate and to another code segment are the same from the program's viewpoint.

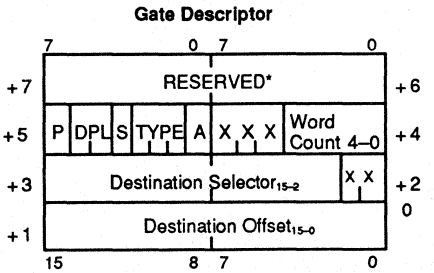
Gates let programs utilize other programs at higher privilege levels, much like a program at an equal privilege level. Programmers do not ever need to differentiate between programs or subroutines that are more privileged than the present program and those that are not. Nevertheless, the system designer may decide to use gates *only* for control transfers that cross privilege levels.

Gates

A gate, a four-word control descriptor, redirects a control transfer to a different code segment in the equal or greater privilege level or to a different task. The four types of gates are call, trap, interrupt, and task gates. The access rights byte differentiates a gate from a segment descriptor, and decides which gate is concerned. The format of a gate descriptor is illustrated in Figure 7-10.

An important feature of a gate is the re-direction it supplies. All four gate types identify a new address that transfers control when induced. A program cannot usually access this destination address. A general protection fault with an error code specifying the invalid selector will occur if the selector is loaded to a call gate into SS, DS, or ES.

Figure 7-10 Gate Descriptor Format



* Must be set to 0 for compatibility with iAPX 386 and future upgrades.

Gate Descriptor Fields

Name	Value	Description
TYPE	4	Call Gate
	5	Task Gate
	6	Interrupt Gate
	7	Trap Gate
P	0	Descriptor Contents are not valid
	1	Descriptor Contents are valid
DPL	0–3	Descriptor Privilege Level
WORD COUNT	0–31	Number of words to copy from callers stack to called procedures stack. Only used with call gate.
DESTINATION SELECTOR	16-bit selector	(Call, Interrupt or Trap Gate) Selector to the target task state segment (Task Gate)
DESTINATION OFFSET	16-bit offset	Entry point within the target Code segment

09729B-012

Just the selector part of an address is used to induce a gate; the offset is disregarded. Concerning the desired function, the program need only know the selector necessary to invoke the gate. The 80C286 will automatically execute at the correct address maintained inside the gate.

Another advantage of a gate is that it supplies a fixed address for any program to activate another program. The calling program's address stays unchanged, even if the destination program's entry address is changed. Therefore, gates furnish a fixed set of entry points that let a task access operating system functions like simple subroutines, but the task is not allowed to simply jump into the middle of the operating system.

The next section discusses call gates, which are used for control transfers within a task that must either be transparently redirected or that require a higher privilege level. A call gate usually identifies a subroutine at a higher privilege level, and the called routine returns through a return instruction. Call gates also maintain delayed binding (resolving target routine addresses at run-time instead of program-generation-time).

Trap and interrupt gates manage interrupt operations that are to be serviced within the present task. Unlike trap gates, interrupt gates cause interrupts to be disabled. A return through the interrupt return instruction is necessary for both trap and interrupt gates.

Task gates control transfers between tasks and use task state segments for task control and status information.Every privilege level in the 80C286 protection model has its own stack. Thus, a new stack is activated when a control transfer (call or return) alters the privilege level.

CALL GATES

Call and jump instructions use call gate descriptors just like a code segment descriptor. The hardware automatically knows that the destination selector references a gate descriptor. The operation of the instruction is then enlarged as decided by the call gate's contents. A jump instruction may access a call gate *only if* the target code segment is at an equal privilege level, while a call instruction utilizes a call gate for equal or greater privilege access.

A call gate descriptor may be in the GDT or the LDT; it may not be in the IDT. The entire layout of a call gate descriptor is shown in Figure 7-10.

Both the long JMP and CALL instructions can reference a call gate. From the standpoint of the program performing a JMP or CALL instruction, it is not obvious that the destination was reached through a call gate and not directly from the instruction's destination address.

Described below are the protection checks executed while transferring control (with the CALL instruction) through a call gate:

- Confirming that access to the call gate is permitted. One of the protection features call gates supply is the access checks, which determine if the call gate may be used (check if the privilege level of the calling program is sufficient).
- Deciding upon the destination address and whether a privilege transition is necessary. With this feature, privilege transitions are clear to the caller.
- Executing the privilege transition, if needed.

Confirming access to a call gate is alike for any call gate and is unaffected by either a JMP or CALL instruction. The rules of privilege that decide whether a data segment may be accessed are used to check if a call gate may be jumped-to or called. With the absence of a call gate, therefore, privileged subroutines can remain hidden from untrusted programs.

The gate's privilege and presence are checked whenever an inter-segment CALL or JMP instruction chooses a call gate. The gate's DPL (in the access byte) is verified against the EPL (MAX (task CPL, selector RPL)). If EPL > CPL, the program is not as privileged as the gate, and thus, it may not change privilege. A general protection fault results in this situation, with an error code specifying the gate. In other instances, the

gate is accessible from the program performing the call, and the control transfer may continue. The descriptor presence is verified following the privilege checks. The Not Present fault results with an error code specifying the gate if the current bit of the gate access rights byte is 0 (i.e., the target code segment is not present).

As shown in Table 7-3, the checks are applied to the call gate's contents. The exception shown occurs if any are violated, and the low order two bits of the error code equal zero for these exceptions.

Table 7-3 **Call Gate Checks**

Type of Check	Fault*	Error Code
Selector is not Null	GP	0
Selector is within Descriptor Table Limit	GP	Selector ID
Descriptor is a Code Segment	GP	Code Segment ID
Code Segment is Present	NP	Code Segment ID
Nonconforming Code Segment DPL > CPL	GP	Code Segment ID

* GP = General Protection; NP = Not-Present Exception.

INTRA-LEVEL TRANSFERS VIA CALL GATE

If the destination code segment is at the equal privilege level as CPL, the transfer is intra-level. The code segment can be non-conforming with DPL = CPL, or conforming, with DPL ≤ CPL. The gate's 32-bit destination address is then loaded into CS:IP.

A General Protection fault with an error code of 0 will result if the IP value is not inside the code segment's limit. The return address is stored in the regular manner if a CALL instruction is used. As a result of the call gate, a different address is placed into CS:IP than that identified in the destination address of the JMP or CALL instruction. This aspect aids systems that require a fixed address to be supplied to programs, although the routine's entry address may change because of various functions, software changes, or segment relocation.

INTER-LEVEL CONTROL TRANSFER VIA CALL GATE

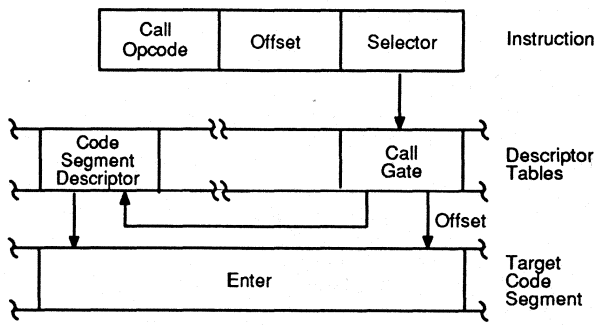
If the destination code segment of the call gate is not at the same privilege level as the CPL, an inter-level transfer is being sought. A General Protection fault with an error code specifying the destination code segment will result, however, if the destination code segment DPL > CPL.

The gate assures that all transitions to a higher privilege level will go to a valid entry point instead of possibly into the middle of a procedure or (even worse) into the middle of an instruction (see Figure 7-11).

Calls to higher privilege levels may be executed through call gates only. It is not possible for a JMP instruction to invoke a privilege change. A General Protection fault with an error code specifying the gate will occur if any attempt is made to use a call gate in this way. Returns to higher privilege levels are also not allowed. Chapter 9 describes how inter-level transitions, due to interrupts, use a different gate.

The RPL field of the CS selector stored as a portion of the return address will always specify the caller's CPL, information necessary to successfully return to the caller's privilege level during the return instruction. Since the CALL instruction sets the CS value on the higher privilege stack, and JMP instructions can never switch privilege levels, it is not possible for a program to arbitrarily set an incorrect return address on the caller's stack.

Figure 7-11 **Call Gate**



STACK CHANGES CAUSED BY CALL GATES

Every privilege level has an individual stack, in order to preserve system integrity. Moreover, every task normally uses individual stacks from additional tasks for every privilege level. These stacks guarantee adequate stack space to process calls from lower privilege levels. Trusted programs may not work properly without them, especially if the calling program does not have adequate space on the caller's stack.

A new stack is chosen, decided upon by the new CPL, when a call gate is used to switch privilege levels. The new stack pointer value is loaded from the Task State Segment (TSS), and the privilege level of the new stack data segment must be the same as the new CPL. If they are not equal, Task Stack fault results, with the saved machine state pointing at the CALL instruction and the error code specifying the incorrect stack selector.

The new stack should include sufficient space to hold the old SS:SP, the return address, and all parameters and local variables necessary to process the call. Strictly read only values, the first stack pointers for privilege levels 0–2 in the TSS are never altered during execution.

The usual practice for passing parameters to a subroutine is to set them onto the stack. To make privilege transitions clear to the called program, a call gate indicates that parameters are to be duplicated from the old stack to the new stack. The word

count field in a call gate (see Figure 7-10) tells how many words (as many as 31) are to be duplicated from the caller's stack to the new stack. No parameters are duplicated if the word count is zero.

Before duplicating the parameters, the new stack is checked to make sure that there is sufficient space to hold the parameters. If not, the result is a stack fault with an error code of 0. Following the duplication of the parameters, the return link is on the new stack (that is, a pointer to the old stack is set in the new one). Specifically, SS:SP points at the return address. Figure 7-12's illustration of the call and return example shows the stack contents following a successful inter-level call.

The caller's stack pointer is stored above the caller's return address as the first two words pushed onto the new stack. The caller's stack, however, may only be maintained for calls to procedures at privilege levels 2, 1, and 0. The level 3 stack will never include links to other stacks because it is not possible for any procedure at any other privilege level to call level 3.

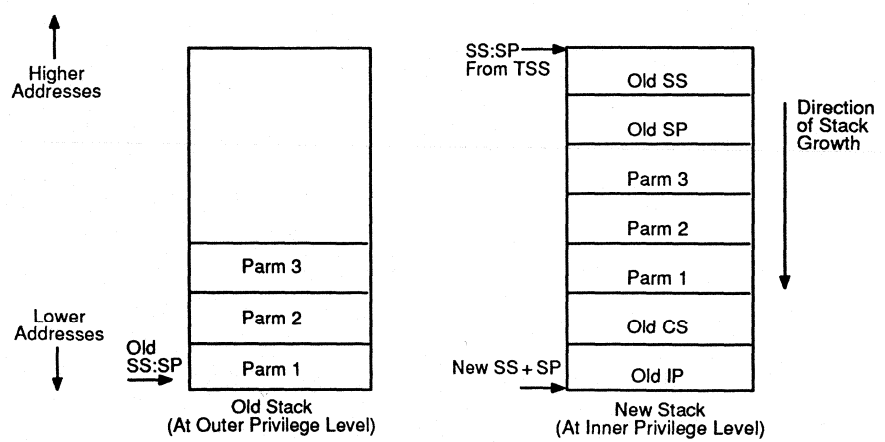
The stored SS:SP link must be used to access all parameters past the last word duplicated for procedures that need over 31 words to call parameters from another privilege level.

The values of the words duplicated onto the new stack are not checked by the call gate, but the called procedure should examine each parameter for validity. Use of the ARPL, VERR, VERW, LSL, and LAR instructions to verify pointer values is described in the section on Pointer Validation in Chapter 11.

Inter-Level Returns

An inter-segment return instruction can also switch levels, but only toward programs of the same or less privilege (in other words, when code segment DPL is numerically greater than or equal to the CPL). The privilege level at which to resume the calling program's performance is specified by the RPL of the selector popped off the stack by the return instruction.

Figure 7-12 Stack Contents after an Inter-Level Call



An inter-level return results whenever the RET instruction meets a saved CS value whose RPL > CPL. The checks performed during such a return are shown in Table 7-4.

The old SS:SP value is then modified by the number of bytes specified in the RET instruction and loaded into SS:SP, but the new SP value is not examined for validity. If SP is incorrect, it is not acknowledged until the first stack operation, and the SS:SP value of the returning program is not maintained. (Note: This value is usually identical to that saved in the TSS.)

The final step in the return is examining the contents of the DS and ES descriptor register. The segment registers are loaded with the null selector if DS or ES refer to segments whose DPL is greater than the new CPL (not including conforming code segments). Any further memory reference that tries to use the segment register with the null selector will result in a general protection fault, and less privileged code will be prevented from accessing higher privilege data accessed earlier by the higher privilege program.

Table 7-4 Inter-Level Return Checks

Type of Check	Exception*	Error Code
SP is not within segment limit	SF	0
SP + *N* + 7 is not in segment limit	SF	0
RPL of Return CS is greater than CPL	GP	Return CS ID
Return CS selector is not null	GP	Return CS ID
Return CS segment is within Descriptor table limit	GP	Return CS ID
Return CS descriptor is a code segment	GP	Return CS ID
Return CS segment is present	NP	Return CS ID
DPL of Return Non-Conforming Code segment = RPL of CS	GP	Return CS ID
SS selector at SP + *N* + 6 is not null	SF	Return SS ID
SS selector at SP + *N* + 6 is within Descriptor Table Limit	SF	Return SS ID
SS descriptor is Writable Data segment	SF	Return SS ID
SS segment is present	SF	Return SS ID
SS segment DPL = RPL of CS	SF	Return SS ID

*SF = Stack Fault, GP = General Protection Exception, NP = Not-Present Exception.

CHAPTER 8

TASKS AND STATE TRANSITIONS

INTRODUCTION

An 80C286 task is one continuous thread of execution, and each task can be separated from the rest of the tasks. Although there may be many tasks affiliated with an 80C286 CPU, only a single task performs at any time. Changing the CPU from performing one task to performing another can result from either an interrupt or an inter-task CALL, JMP or IRET. Each task is identified by a hardware-acknowledged data structure.

The 80C286 supplies a high performance task switch operation with total separation between tasks. Only 22 μs at 8 MHz (18 μs at 10 MHz) are needed to perform a full task-switch operation. High-performance, interrupt-driven, multi-application systems needing the advantages of protection are possible with the 80C286.

The following benefits emerge from the 80C286 task switch:

- Quicker task switch. A task switch is a single instruction that microcode executes. This scheme is 2–3 times quicker than a direct task switch instruction. A quick task switch results in a substantial performance boost for heavily multi-tasked systems beyond ordinary methods.
- More dependable, adaptable systems. The separation between tasks and the high speed task switch lets other tasks handle interrupts instead of the presently interrupted task. This separation of interrupt handling code from usual programs inhibits unwanted interactions between them. Adding an interrupt handler is as sound and simple as adding a new task, so the interrupt system can become more adaptable.
- Each task is protected from all others by the separation of address spaces discussed in Chapter 7, such as appropriation of distinct stacks to each active privilege level in each task (unless direct sharing is provided in advance). A single task cannot affect another task's data if the address spaces of the two tasks contain no shared data. Because code segments are never writable, code sharing is secure all the time.

TASK STATE SEGMENTS AND DESCRIPTORS

A special control segment called a Task State Segment (TSS) identifies tasks, and for every task, an individual TSS must exist. A task, by definition, is comprised of its address space and execution state. Inter-segment Jump or Call instructions (whose destination address refers to a task state segment or a task gate) activate a task.

The Task State Segment (TSS) contains a special descriptor, and the Task Register inside the CPU includes a selector to that descriptor. Each TSS selector value is distinct, supplying an explicit identifier for each task. An operating system, therefore, can use the TSS selector's value to distinctly specify the task.

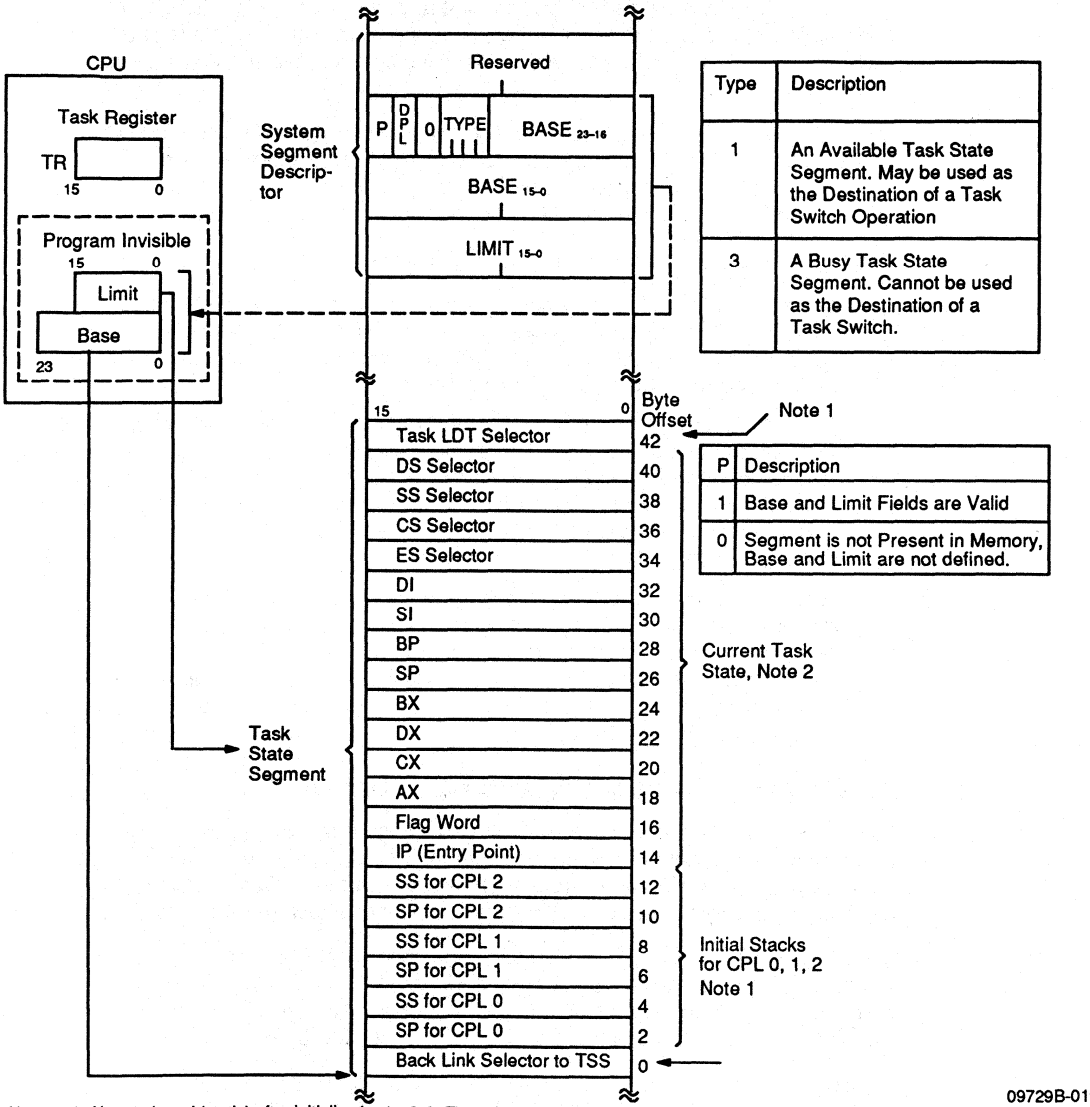
A TSS includes 22 words that identify the contents of all registers and flags, the initial stacks for privilege levels 0–2, the LDT selector, and a link to the TSS of the previ-

ously performing task. The layout of the TSS, which is not writable like a regular data segment, is shown in Figure 8-1.

Each TSS is made of a static portion and a dynamic portion. The 80C286 never changes the static entries, but the dynamic entries are modified by every task switch out of this task. The task LDT selector and the original SS:SP stack pointer addresses for levels 0–2 comprise the static parts of this segment.

The changeable or dynamic portion of the task state segment includes all dynamically-variable and programmer-visible processor registers, such as flags, segment

Figure 8-1 Task State Segment and TSS Registers



09729B-019

Notes: 1. Never altered (static) after initialization by O.S. The values as initialized for this task are always valid SS:SP values to use upon entry to that privilege level (0, 1, or 2) from a level of lesser privilege.
2. Changed during task switch.

registers, and the instruction pointer. This portion also contains the linkage word that chains nested activations of various tasks.

A history of which tasks activated others is furnished by the link word. The link word is significant for resuming an interrupted task when the interrupt has been serviced. Since the TSS is not writable by the interrupt task, placing the back link in the TSS shields the identity of the interrupted task from modifications by the interrupt task. (In the majority of systems, only the operating system has enough privilege to produce or use a writable data segment alias descriptor for the TSS.)

In the TSS, the stack pointer entries for privilege levels 0–2 are static (i.e., they are never written during a privilege or task switch). They identify the stack that is to be used when entering that privilege level. The operating system initializes these stack entries when the task is generated. No stack needs to be appropriated for a privilege level that is never used.

When entering a higher privilege level, the caller's stack pointer is not stored in the TSS. Instead, it is stored on the stack of the new privilege level. Exiting the privilege level calls for popping the caller's return address and stack pointer off the present stack. The stack pointer at that time will be identical to the initial value loaded from the TSS when the privilege level was entered.

The only stack active at any time is the one defined by the SS and SP registers. Those stacks at outer (less privileged) levels that called the present level are the only other stacks that may be non-empty. Stacks at inner levels must be empty, because outward (to numerically greater privilege levels) calls from inner levels are prohibited.

The stack pointer's location for an outer privilege level will always be at the beginning of the stack of the inner privilege level called by that level, and that stack may be the original stack for this privilege level or an outer level. Examine the beginning of the stack for this privilege level. The beginning stack address for levels 0–2 are included in the TSS. If the RPL of the stored SS selector is the necessary privilege level, then the stack pointer has been located. In any other case, examine the stored SS:SP value at the start of the stack identified by that value.

Task State Segment Descriptors

A special descriptor, which must always be accessible, is used for task state segments. Thus, it can be seen only in the GDT. The access byte differentiates TSS descriptors from data or code segment descriptors. The descriptor is for a TSS when bits 0 through 4 of the access byte are 00001 or 00011.

Figure 8-2 illustrates the entire layout of a task state segment descriptor.

The descriptor, like a data segment, includes a base address and limit field. The limit must be at least 002BH (43) to include the least amount of information a TSS requires. If an attempt is made to change to a task whose TSS descriptor limit is less than 43, an invalid task exception results, and the error code defines the faulty TSS.

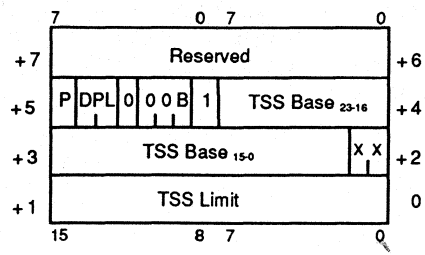
The P-bit (Present) flag reveals whether this descriptor includes presently valid information, with 1 meaning yes and 0, no. A Not Present exception code specifying the task state segment selector results when a task switch tries to reference a not-present TSS.

The descriptor privilege level (DPL) directs use of the TSS by JMP or CALL instructions. Like the reasoning behind call gates, the DPL can stop a program from calling the TSS (thereby causing a task switch).

Because TSS is a control segment descriptor, bit 4 is always 0. SS, DS, and ES cannot access control segments. A general protection trap results if any attempt is made to load those segment registers with a selector that refers to a control segment. This rule will not let the program improperly change a control segment's contents.

Bit 1 of the access byte indicates whether TSS descriptors are in one of two states: idle or busy. Since tasks are not re-entrant, this definition is necessary. A busy TSS may not be activated.

Figure 8-2 **TSS Descriptor**



B = 1 means task is busy and not available.

TASK SWITCHING

Listed below are the four ways a task switch may occur:

1. The destination selector of a long JMP or CALL instruction references a TSS descriptor. The offset portion of the destination address is disregarded.
2. When the NT bit in the flag word = 1, an IRET instruction is performed. The new task TSS selector is found in the present TSS's back link field.
3. The destination selector of a long JMP or CALL instruction references a task gate. The offset portion of the destination address is disregarded, and the new task TSS selector is in the gate.
4. An interrupt occurs. This interrupt's vector references a task gate in the interrupt descriptor table, and the new task TSS selector is in the gate.

A task switch operation needs no new instructions. This function is executed by the standard 8086 JMP, CALL, IRET, or interrupt operations. What distinguishes the standard instruction from a task switch is either the type of descriptor referenced (for CALL, JMP, or INT) or the NT bit (for IRET) in flag word.

Using the CALL or INT instruction to change tasks suggests a return is expected from the called task, while use of the JMP and IRET instructions suggests no return is expected from the new task.

The IRET instruction invokes a return to the task that called the present one through CALL or INT instruction, whenever NT = 1.

The rules of privilege level limit access to TSS and task gate descriptors. The data access rules are utilized; hence, task switches may be limited to programs of adequate privilege. Address space isolation is not applicable to TSS descriptors, because they must appear in the GDT.

The task switch operation includes the eight steps described below:

1. Sanction the requested task switch. For a task switch requested through a JMP, CALL, or INT instruction, check that the present task can change to the requested task. The gate's DPL or the TSS descriptor for the requested task must be more than or equal to the requesting task's CPL and RPL. If these conditions are not met, a General Protection fault (#13) results with an error code specifying the descriptor (i.e., the gate selector if the task switch is requested through a task gate, or the selector for the TSS if the task switch is requested through a TSS descriptor).

   These checks are not executed if a task switch results from an IRET instruction.

2. Check that the new TSS is present and that the new task is accessible (Not Busy). A Not Present exception (#11) is indicated if the new TSS descriptor is stamped Not Present (P = 0). A General Protection exception (#13) occurs if the new TSS is stamped Busy.

   The task switch operation actually starts now and a complete confirmation of the new TSS is performed. Table 8-1 lists the conditions that may disqualify the new TSS, along with the exception that occurs and the error code that is pushed on the stack for every case. These tests are executed at varying points during the course of the rest of the steps of the task switch operation that follow.

3. Stamp the new task as BUSY by setting the BUSY bit in the new TSS descriptor to 1.

4. Maintain the dynamic part of the old TSS and load TR with the new TSS's selector, base, and limit. Set all CPU registers (except DS, ES, CS, SS, and LDT) to comparable values from the new TSS.

5. Set the Nested Task (NT) flag in the new TSS to 1, if nesting tasks. Also, set the CPU flag register's Task Switched flag (TS) to 1.

6. Sanction the new TSS's LDT selector and the LDT descriptor. With the LDT descriptor, load the LDT cache (LDTR).

7. Sanction the new TSS's SS, CS, DS, and ES fields and load these values in their respective caches (SS, CS, DS, and ES registers).

8. Sanction the new TSS's IP field and then begin performing the new task from CS:IP.

Appendix B (80C286 Instruction Set) provides a more detailed explanation of steps 3–5 under a quasi procedure SWITCH_TASKS. Note how the exceptions Table 8-1 described may actually happen during a task switch. The pseudo code description of Appendix B's 286 instructions CALL, JMP, INT, and IRET also further explains the exceptions that may result during steps 1, 2, and 8. Such information can be extremely handy when debugging any protected mode code.

Notice that the outgoing task's state is constantly saved. If that task's performance is resumed, it will begin following the instruction that caused the task switch. The registers' values will be identical to the values when the task was no longer running.

All task switches place the Task Switched (TS) bit in the Machine Status Word (MSW). Whenever processor extensions like the AMD 80C287 device are present, this flag is used. The TS bit indicates that the processor extension's context may not be part of the current 80C286 task.

Validity tests on a selector make sure the selector is in the appropriate Table (that is, the LDT selector references GDT), within the bounds of the table, and referring to the appropriate descriptor type (i.e., the LDT selector references the LDT descriptor).

Notice that every register of the new task is loaded between steps 3 and 4 in Table 8-1. There may be multiple protection rule violations in the contents of the new segment register. If an exception arises in the context of the new task because of checks executed on the newly loaded descriptors, the DS and ES segments may not be available, even though the segment registers have non-zero values. These selector values must be kept for subsequent reuse. Unless the exception handler examines them beforehand and solves any potential problems, another protection exception may result when the exception handler loads these segment registers again.

A task switch provides for flexibility in the outgoing and incoming tasks' privilege levels. The outgoing task's privilege level does not restrict the privilege level at which performance resumes in the incoming task. Since both tasks are separated from each other with different address spaces and machine states, this is rational. The privilege rules do not allow inappropriate access to a TSS. The only interaction between the tasks concerns the fact that one started the other, and the incoming task may resume the outgoing task by performing an IRET instruction.

Table 8-1 **Checks Made during a Task Switch**

	Test	Exception*	Error Code
1	Incoming TSS descriptor is present	NP	Incoming TSS selector
2	Incoming TSS is idle	GP	Incoming TSS selector
3	Limit of incoming TSS greater than 43	Invalid TSS	Incoming TSS selector
4	LDT selector of incoming TSS is valid	Invalid TSS	LDT selector
5	LDT of incoming TSS is present	Invalid TSS	LDT selector
6	CS selector is valid	Invalid TSS	Code segment selector
7	Code segment is present	NP	Code segment selector
8	Code segment DPL matches CS RPL	Invalid TSS	Code segment selector
9	Stack segment is valid	SF	Stack segment selector
10	Stack segment is writable data segment	GP	Stack segment selector
11	Stack segment is present	SF	Stack segment selector
12	Stack segment DPL = CPL	SF	Stack segment selector
13	DS/ES selectors are valid	GP	Segment selector
14	DS/ES segments are readable	GP	Segment selector
15	DS/ES segments are present	NP	Segment selector
16	DS/ES segment DPL ≥ CPL if not conform	GP	Segment selector

*NP = Not-Present Exception, GP = General Protection Fault, SF = Stack Fault

TASK LINKING

Back Link,(a field of the TSS) includes the selector of the TSS of a task to be resumed when the present task finishes. The interrupted task's TSS selector automatically writes to the back link field of an interrupt-initiated task.

Originated by a CALL instruction, a task switch points the back link at the outgoing task's TSS as well. This task nesting is shown to programs through the Nested Task (NT) bit in the incoming task's flag word.

Task nesting is required if interrupt functions are to be processed as individual tasks. The interrupt function is thus kept separate from all other system tasks. To resume the interrupted task, the interrupt handler performs an IRET instruction similar to that of an 8086 interrupt handler, and the IRET instruction then generates a task switch to the interrupted task.

A task is completed when the IRET instruction is performed with the NT bit in the flag word set. The NT bit is automatically set/reset by task switch operations as applicable. Performing an IRET instruction with NT cleared invokes the regular 8086 interrupt return function to be executed. No task switch results.

A task switch to the task identified by the back link field of the current TSS occurs by performing IRET with NT set. The selector value is obtained and confirmed as pointing to a valid, available TSS. Then the regular task switch operation results. Following the task switch's completion, the outgoing task, now idle, is considered available to process another interrupt.

The effect of task switch operations caused by JMP, CALL, or IRET instructions on the busy bit, NT bit, and link word of the incoming and outgoing task is indicated in Table 8-2.

A General Protection fault occurs, with the saved machine state looking as if the instruction had not performed, if any of the busy bit requirements indicated in Table 8-2 are violated. The TSS with the Busy bit is designated by the resulting error code.

A bus lock used during the testing and setting of the TSS descriptor busy bit guarantees that two processors do not activate the same task simultaneously.

The linking order of tasks may have to be altered to resume an interrupted task before completion of the task that interrupted it. To withdraw a task from the list, trusted operating system software must first modify the back link field in the TSS of the interrupting task, and then clear the busy bit in the TSS descriptor of the task withdrawn from the list.

When trusted software removes the link between two tasks, it should set a value in the back link field, which will forfeit control to that trusted software when the task tries to restart performance of another task through IRET.

Table 8-2 **Effect of a Task Switch on BUSY and NT Bits and the Link Word**

Affected Field	JMP Instruction Effect	CALL/INT Instruction Effect	IRET Instruction Effect
Busy bit of incoming task TSS descriptor	Set, must be 0 before	Set, must be 0 before	Unchanged, must be set
Busy bit of outgoing task TSS descriptor	Cleared	Unchanged (will already be 1)	Cleared
NT bit in incoming task flag word	Cleared	Set	Unchanged
NT bit in outgoing task flag word	Unchanged	Unchanged	Cleared
Back link in incoming task TSS	Unchanged	Set to outgoing task TSS selector	Unchanged
Back link of outgoing task TSS	Unchanged	Unchanged	Unchanged

TASK GATE

Several events can activate a task. To support this need, task gates are supplied. Task gates are utilized in the same manner as call and interrupt gates. The end effect of jumping to or calling a task gate is identical to jumping to or calling directly to the task gate's TSS. A task gate's layout is shown in Figure 8-3.

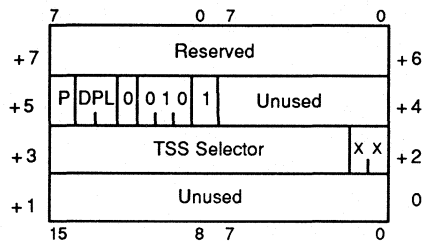
The access byte field being 00101 in bits 0 through 4 specifies a task gate, which supplies an added level of indirection between the destination address and the TSS selector value. The JMP or CALL destination address' offset portion is disregarded.

Gate use furnishes flexibility in managing task access. Task gates can be found in the GDT, IDT, or LDT, but the TSS descriptors for all tasks must reside in the GDT. They are usually set at level 0 to stop any task from inappropriately activating another task. Task gates set in the LDT permit private access to particular tasks with complete privilege control.

The data segment access rules apply to task gate accessing through JMP, CALL, or INT instructions. The destination selector's effective privilege level (EPL) must be numerically less than or the same as the task gate descriptor's DPL. A General Protection fault with an error code specifying the task gate involved will occur if this requirement is violated.

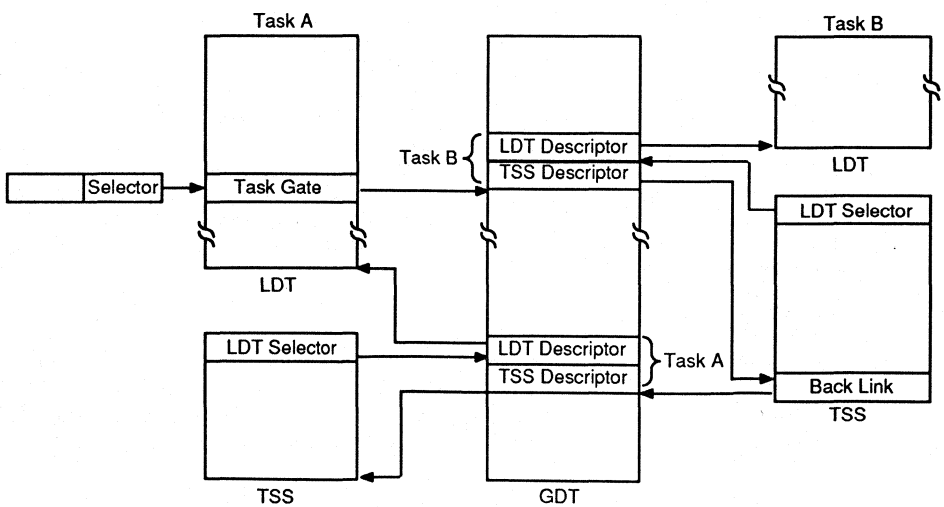
The TSS selector from the gate is read as soon as access to the task gate has been confirmed, and the RPL of the TSS selector is disregarded. From now on, all the checks and actions executed for a JMP or CALL to a TSS following confirmation of access are executed. An example of a task switch through a task gate is illustrated in Figure 8-4.

Figure 8-3 **Task Gate Descriptor**



* Must be set to 0.

Figure 8-4 **Task Switch Through a Task Gate**



14554A-045

CHAPTER 9

INTERRUPTS AND EXCEPTIONS

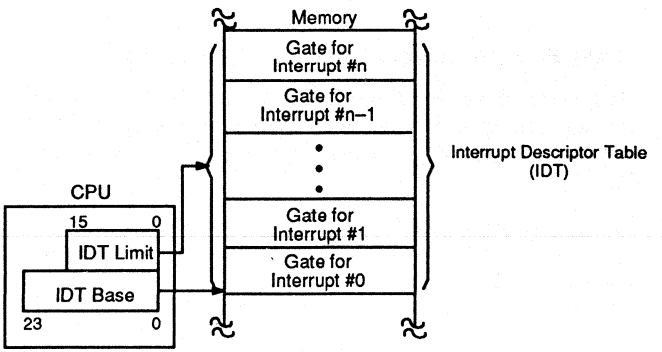
Interrupts and exceptions are particular instances of control transfer in a program. An interrupt is the result of an event that is unconnected to the presently executing program. An exception is a direct result of the presently executing program. There are two types of interrupts: external and internal. External interrupts are produced by either the INTR or NMI input pins, while the INT instruction results in internal interrupts. An exception is the result when an instruction cannot be performed normally. Their causes differ, but interrupts and exceptions utilize the same control transfer techniques and privilege rules. Thus, the term interrupt will also apply to exceptions in the discussions below.

The program that services an interrupt may perform in the context of the task that caused the interrupt (uses the identical TSS, LDT, stacks, etc.) or may be an individual task. The selection is dependent on the function to be executed and the level of separation needed.

INTERRUPT DESCRIPTOR TABLE

An interrupt may be caused by multiple and various events. Every interrupt source is assigned a number called the interrupt vector, to allow the reason for an interrupt to be easily determined. As many as 256 separate interrupt vectors (numbers) are possible (see Figure 9-1).

Figure 9-1 **Interrupt Descriptor Table Definition**



The IDT may contain interrupt gates, traps or task gates only.

09729B-017

A table identifies each interrupt vector's handler. The Interrupt Descriptor Table (IDT) identifies the interrupt handlers for as many as 256 separate interrupts. The IDT, which is in physical memory, is pointed to by the contents of the on-chip IDT register containing a 24-bit base and a 16-bit limit. The IDTR is usually loaded with the LIDT instruction by code that performs at privilege level 0 in the course of system initialization. The IDT may be found anywhere in the 80C286's physical address space.

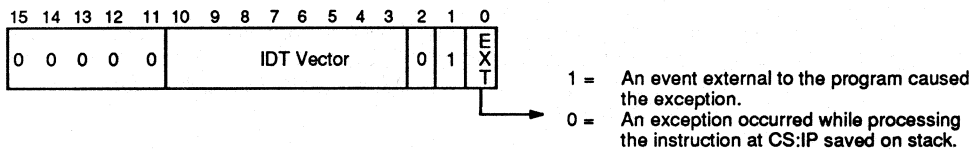
Every IDT entry is a 4-word gate descriptor with a pointer to the handler. There are three types of gates allowed in the IDT: interrupt gates, trap gates, and task gates. Interrupt and task gates process interrupts within one task. Task gates activate a task switch. An exception will occur if any other descriptor type in the IDT is referred to by an interrupt.

The IDT is not required to contain all 256 entries; less than the full number of entries are supported by a 16-bit limit register. Setting a zero in the access rights byte indicates unused entries. A General Protection fault will occur with an error code specifying the invalid interrupt vector pushed on the stack, if an attempt is made to access an entry beyond the table limit, or if the incorrect descriptor type is located (see Figure 9-2).

Exception error codes referring to an IDT entry can be defined by bit 1 of the set error code. If an event external to the program caused the interrupt (i.e., an external interrupt, a single step, a processor extension error, or a processor extension not present), bit 0 of the error code equals 1.

Interrupts 0–31 are reserved, but some of the interrupts are used for instruction exceptions. The minimum IDT limit must be 255 ($32 \times 8 - 1$) to provide for the least number of interrupts. The 224 interrupts that remain are accessible to the user.

Figure 9-2 IDT Selector Error Code



HARDWARE-INITIATED INTERRUPTS

Some external event that invokes either the INTR or NMI input pins of the processor causes hardware-initiated interrupts. Events using the INTR input are considered maskable interrupts, while events using the NMI input are considered non-maskable interrupts.

All 224 user-specified interrupt sources use the INTR input, but each source is capable of using an individual interrupt handler. An 8-bit vector provided by the interrupt controller specifies the particular interrupt. The processor executes the interrupt acknowledge bus sequence in order to read the interrupt ID.

Software can inhibit maskable interrupts (from the INTR input) by setting the interrupt flag bit (IF) to 0 in the flag word. The IF bit does not affect exceptions or interrupts that the INT instruction caused nor does it affect processor extension interrupts.

The type of gate placed into the IDT for the interrupt vector determines whether additional maskable interrupts stay enabled while that interrupt is being serviced. The flag word that was stored on the stack reveals the maskable interrupt enable status of the processor before the interrupt. By resetting the IF flag, the procedure servicing a maskable interrupt can also impede additional maskable interrupts in the course of its work.

The NMI input is the source of non-maskable interrupts, which have a greater priority than the maskable interrupts (i.e., the non-maskable interrupt will be serviced first in case of simultaneous requests). A non-maskable interrupt has a fixed vector (#2) and, consequently, a bus interrupt acknowledge sequence is not necessary. A normal use of an NMI is to activate a procedure to manage a power failure or some other serious hardware exception.

Until an IRET instruction is performed, a procedure servicing an NMI will no longer be interrupted by other non-maskable interrupt requests. The hardware stores a further NMI request (just one NMI request can be remembered), which will be serviced following the first IRET instruction. In order to deter a maskable interrupt from interrupting the NMI interrupt handler, the IF flag should be cleared in one of two ways: by using an interrupt gate in the IDT or by setting IF = 0 in the flag word of the task involved.

SOFTWARE-INITIATED INTERRUPTS

Software-initiated interrupts, which are not maskable, occur directly as interrupt instructions or as the result of an unusual condition that deters program execution from continuing. There are two interrupt instructions that directly cause an interrupt: INT n and INT 3. INT n permits identification of any interrupt vector; INT 3 reflects interrupt vector 3 (Breakpoint).

Other instructions, such as INTO, BOUND, DIV, and IDIV, may invoke an interrupt, but this depends on the overflow flag or the operand values. These instructions have already specified vectors associated with them in the first 32 interrupts reserved.

A whole interrupt group, called exceptions (which are non-maskable), is used to spot faults or programming errors (in operands or privilege levels). Exceptions have fixed vectors within the first 32 interrupts, as well. Many of these exceptions pass an error code on the stack, but other interrupt types do not. These error codes, as well as the priority among interrupts that occur simultaneously are discussed later in this chapter.

INTERRUPT GATES AND TRAP GATES

Interrupt gates and trap gates are particular descriptor types that may only be found in the interrupt descriptor table. Whether or not the interrupt enable flag is to be cleared determines the difference between a trap and an interrupt gate. An interrupt gate is for a procedure that enters with interrupts disabled (the interrupt enable flag cleared), and entry through a trap gate does not alter the interrupt enable status. Whenever an interrupt makes use of these gates, the NT flag is always cleared (after the old NT state is stored on the stack). Interrupts with either of the gates in the corresponding IDT entry will be addressed in the present task.

The structure of interrupts and trap gates is the same as that of the call gates described earlier. The gate includes the selector and entry point for a code segment to manage the interrupt or exception (see Figure 9-3).

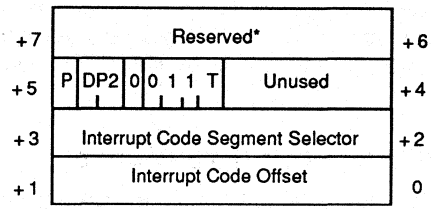
The Present bit, the descriptor privilege level, and the type identifier are included in the access byte. Bits 0–4 of the access byte have a value of 00110 for interrupt gates and 00111 for trap gates, whereas neither of these gates use byte 5 of the descriptor. Byte 5 is used solely by the call gate for the parameter word-count.

When passing control to a non-conforming code segment, trap and interrupt gates permit privilege level transition. The DPL of the chosen target code segment desig-

nates the new CPL, like a call gate, and the DPL of the new non-conforming code segment must be numerically less than or equal to CPL.

If the new code segment is conforming, no privilege transition results. A General Protection exception will occur if the conforming code segment's DPL is more than the CPL.

Figure 9-3 Trap/Interrupt Gate Descriptors



* Must be set to 0
T = 1 for trap gate.
T = 0 for interrupt gate.

Like all descriptors, these gates in the IDT bear a privilege level. The DPL regulates access to interrupts with the INT n and INT 3 instructions. The program CPL must be less than or equal to the gate DPL for access. If this is not the case, a General Protection exception will occur with an error code specifying the chosen IDT gate. In the case of exceptions and external interrupts, the program CPL is disregarded while accessing the IDT.

Interrupts that use a trap or an interrupt gate are treated like an 8086 interrupt. The flags and return address of the interrupted program are stored on the interrupt handler's stack. The interrupt handler performs an IRET instruction to go back to the interrupted program.

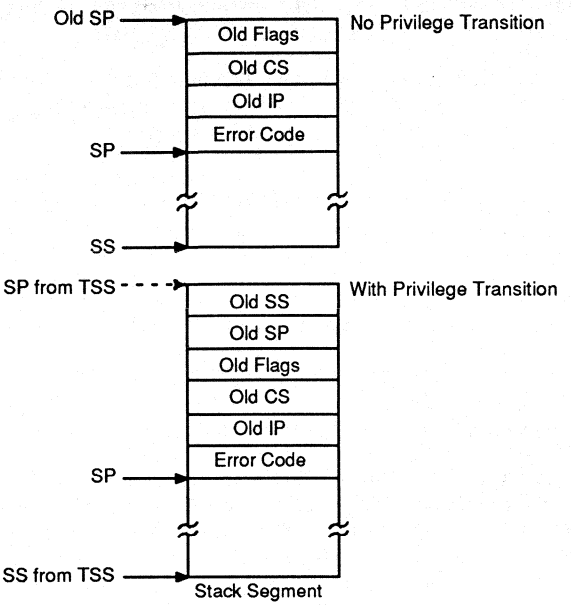
If a higher privilege is necessary to handle the interrupt, the TSS will provide a new stack. The old privilege level's stack pointer will also be stored on the new stack like a call gate. The stack contents following an exception with an error code are illustrated in Figure 9-4, with and without a privilege level change.

The error code will be pushed onto the new stack following the return address (see Figure 9-4) if an Interrupt or Trap gate handles the exception. The error code is pushed onto the new task's stack, if a task gate is used, and the return address is stored in the old TSS.

If an interrupt gate handles an interrupt, it is presumed that the chosen code segment has adequate privilege to re-enable interrupts. If CPL is numerically greater than IOPL, the IRET instruction will not re-enable interrupts.

The checks executed during an interrupt operation using an interrupt or trap gate are shown in Table 9-1. When an event external to the program is concerned, EXT equals 1. Otherwise, EXT equals 0. External events include the following: maskable or non-maskable interrupts, single step interrupt, processor extension segment overrun interrupt, numeric processor Not Present exception or numeric processor error. The EXT bit indicates that the interrupt or exception does not concern the instruction at CS:IP. Every error code has bit 1 set to signal involvement of an IDT entry.

Figure 9-4 **Stack Layout after an Exception with an Error Code**



14554A-046

Table 9-1 **Trap and Interrupt Gate Checks**

Check	Exception*	Error Code
Interrupt vector is in IDT limit	GP	IDT entry × 8 + 2 + EXT
Trap, Interrupt, or Task Gate in IDT Entry	GP	IDT entry × 8 + 2 + EXT
If INT instruction, gate DPL ≥ CPL	GP	IDT entry × 8 + 2 + EXT
P bit of gate is set	NP	IDT entry × 8 + 2 + EXT
Code segment selector is in descriptor table limit	GP	CS selector × 8 + EXT
CS selector refers to a code segment	GP	CS selector × 8 + EXT
If code segment is non-conforming, Code Segment DPL ≤ CPL	GP	CS selector × 8 + EXT
If code segment is non-conforming, and DPL < CPL and if SS selector in TSS is in descriptor table limit	TS	SS selector × 8 + EXT
If code segment is non-conforming, and DPL < CPL and if SS is a writable data segment	TS	SS selector × 8 + EXT
If code segment is non-conforming, and DPL < CPL and code segment DPL = stack segment DPL	TS	Stack segment selector + EXT
If code segment is non-conforming, and DPL < CPL and if SS is present	SF	Stack segment selector + EXT
If code segment is non-conforming, and DPL ≤ CPL and if there is enough space for 5 words on the stack (or 6 if error code is required)	SF	SS selector + EXT
If code segment is conforming, then DPL ≤ CPL	GP	Code segment selector + EXT
If code segment is not present	NP	Code segment selector + EXT
If IP is not within the limit of code segment	GP	0 + EXT

*GP = General Protection Exception, NP = Not Present Exception, SF = Stack Fault

After the interrupt is serviced, the service routine returns control to the interrupted routine through an IRET instruction. The exception handler must remove the error code, if passed, from the stack prior to executing IRET.

The NT flag is cleared whenever an interrupt that uses an interrupt or trap gate occurs. Performing IRET with NT = 0 results in the usual interrupt return function, while performing IRET with NT = 1 creates a task switch.

Like the RET instruction, unless a task switch happens, IRET must return to a level of equal or lesser privilege. The IRET instruction performs like the inter-segment RET instruction, except the flag word is popped and no stack pointer update for parameters is executed because no parameters are on the stack.

The new CPL (that is, the RPL of the return address CS selector) is compared to the present CPL to identify an inter-level IRET. The IP and flags are popped and performance proceeds, if the two CPLs are equal.

An inter-level return through IRET has all the checks listed in Table 7-4, the only difference being the added word on the stack for the old flag word.

Interrupt gates are generally affiliated with high-priority hardware interrupts for automatically disabling interrupts when they are activated. Since trap gates do not disable the maskable hardware interrupts, they are generally software-activated. Low-priority interrupts (a timer), however, are often activated through a trap gate so that other devices of higher priority can interrupt the handler of that lower priority interrupt.

Interaction of the interrupt enable flag and interrupt type with the particular gate type used is shown in Table 9-2.

Table 9-2 **Interrupt and Gate Interactions**

Type of Interrupt	Type of Gate	Further NMIs?	Further INTRs?	Further Exceptions?	Further Software Interrupts?
NMI	Trap	No	Yes	Yes	Yes
NMI	Interrupt	No	No	Yes	Yes
INTR	Trap	Yes	Yes	Yes	Yes
INTR	Interrupt	Yes	No	Yes	Yes
Software	Trap	Yes	Yes	Yes	Yes
Software	Interrupt	Yes	No	Yes	Yes
Exception	Trap	Yes	Yes	Yes	Yes
Exception	Interrupt	Yes	No	Yes	Yes

TASK GATES AND INTERRUPT TASKS

The 80C286 lets interrupts directly generate a task switch. A task switch results whenever an interrupt vector chooses a task gate entry in the IDT. The error code is pushed onto the stack of the new task, provided a task gate handles that exception.

A task gate presents two benefits over interrupt gates:

1. It automatically stores all of the processor registers as part of the task-switch operation; an interrupt gate stores only the flag register and CS:IP.

2. The new task is entirely separated from the interrupted task. Address spaces are kept separate, and the interrupted task's privilege level does not affect the interrupt-handling task.

Once the TSS selector is obtained from the task gate, an interrupt task switch functions like any other task switch. Privilege and presence rules, as with trap or interrupt gates, are exercised when accessing a task gate during an interrupt.

Interrupts which produce a task switch place the NT bit in the new task's flags, and the TSS selector of the interrupted task is stored in the back link field of the new TSS. Since NT was previously set, the interrupting task performs IRET to execute a task switch so as to resume the interrupted task. The interrupt task state is stored in its TSS prior to restoring control to the interrupted task. NT is returned to its initial value in the interrupted task.

Since the interrupt handler state after performing IRET is stored, a re-entry of the interrupt service task activates the execution of the instruction following IRET. Thus, when the next interrupt occurs, the machine state will be the same as when the IRET instruction was performed.

Note: An interrupt task continues execution every time it is re-activated, while an interrupt procedure restarts execution. Since interrupts occur prior to performance of an instruction, the interrupted task resumes execution at the point of interruption.

An interrupt task must avoid any more interrupts while it is processing. If a task gate referring to a busy TSS is used during an interrupt process, a General Protection exception results. The IF bit in the flag word (stored in the TSS) must equal zero for further interrupts to occur during task performance.

Scheduling Considerations

A software-scheduled operating system must be able to handle interrupts that appear during scheduled tasks and generate a task switch to other tasks. The interrupt-scheduled tasks may call the operating system and finally the scheduler, which must acknowledge that the task that just called it is not the same task last scheduled by the operating system.

An interrupt initiated task switch occurs if the Task Register (TR) does not include the TSS selector of the last scheduled task. It is possible that more than one task was interrupt-scheduled since the scheduler last ran. The scheduler must locate (through the back link fields in every TSS) all interrupted tasks. The scheduler can then clear those links and reset the Busy bit in the TSS descriptors, returning them to the scheduling queue for a new evaluation of execution priorities. Unless the interrupted tasks are returned to the scheduling queue, they would have to wait for a restart via the task that interrupted them.

To find tasks that have been interrupt-scheduled, the scheduler checks the current task's TSS back link (first word of the TSS), which points to the interrupted task. If not the last task scheduled, then that task's back link field in the TSS points to an interrupted task as well.

The scheduler should set the back link field of every interrupt-scheduled task to point to a scheduling task. When the interrupt-scheduled task performs IRET, the scheduling task will then reschedule the highest priority task.

Deciding Between Task, Trap, and Interrupt Gates

Either a trap/interrupt gate or a task gate can manage interrupts and exceptions. The benefits of a task gate are that every register is saved and a new set is loaded with total separation between the interrupted task and the interrupt handler. Quicker response to an interrupt for easy operations and ready access to pointers in the context of the interrupted task are advantages of a trap/interrupt gate. IRET is used by all interrupt handlers to restart the program that was interrupted.

Trap/interrupt gates need for the interrupt handler to be able to perform at the same or higher privilege level than the interrupted program. If any program performing at level 0 can be interrupted via a trap/task gate, the interrupt handler must also perform at level 0 to escape General Protection exception. To permit access from any task, all code, data, and stack segment descriptors must appear in the GDT. Setting all system interrupt handlers at privilege level 0, however, may be inconsistent with upholding the integrity of level 0 programs.

The use of a task gate is necessary for some exceptions. The invalid task state segment exception (#10) can result from errors in the initial TSS and the target TSS. Handling the exception within the same task could result in recursive interrupts or other undesirable effects that are not easy to track. The Double Fault exception (#8) should also use a task gate to avoid shutdown from another protection violation that occurs while the exception is being serviced.

PROTECTION EXCEPTIONS AND RESERVED VECTORS

A protection violation will generate an exception, or a non-maskable interrupt. The task that created such a fault can handle it if an interrupt or trap gate is used, or a different task can handle it if a task gate is used (in the IDT).

Protection exceptions can be divided into program errors and implied service requests. The latter contain stack overflow and Not Present faults. Attempting to write into a read-only segment or violating segment limits are examples of program errors.

Although many different kinds of protection violation use the same General Protection fault vector, service requests may use various interrupt vectors. The reserved exceptions and interrupts are listed in Table 9-3. Interrupts 0–31 are reserved.

Table 9-3 **Reserved Exceptions and Interrupts**

Vector Number	Description	Restartable	Error Code on Stack
0	Divide Error Exception	Yes	No
1	Single Step Interrupt	Yes	No
2	NMI Interrupt	Yes	No
3	Breakpoint Interrupt	Yes	No
4	INTO Detected Overflow Exception	Yes	No
5	BOUND Range Exceeded Exception	Yes	No
6	Invalid Opcode Exception	Yes	No
7	Processor Extension Not Available Exception	Yes	No
8	Double Exception Detected	No	Yes (Always 0)
9	Processor Extension Segment Overrun Interrupt	No	No
10	Invalid Task State Segment	Yes	Yes
11	Segment Not Present	Yes	Yes
12	Stack Segment Overrun or Not Present	Yes	Yes
13	General Protection	Yes*	Yes

*Except for writes into read-only segments.

Table 9-4 shows the fixed order in which external interrupt requests are processed when they occur at the same time. The machine state is saved for every interrupt serviced, and the gate or the TSS provides the new CS:IP. If other interrupts are still enabled, they are processed prior to the first instruction of the present interrupt handler (i.e., the last interrupt processed is the first serviced).

Table 9-4 **Interrupt Processing Order**

Order	Interrupt
1	Instruction exception
2	Single step
3	NMI
4	Processor extension segment overrun
5	INTR

After the exceptional condition is withdrawn, all but two exceptions can be resumed: the processor extension segment overrun and writing into read-only segments with XCHG, ADC, SBB, RCL, and RCR instructions. The return address typically points to the failing instruction, which includes all leading prefixes.

The processor extension status registers include the instruction and data addresses for the processor extension segment overrun.

Interrupt handlers for the majority of exceptions are assigned an error code that defines the selector involved, or a 0 in bits 15–3 of the error code field if no selector is involved. The error code is pushed last, following the return address, on the stack that will be operational when the trap handler starts to execute. Thus, to locate the error code, the handler will not have to access another stack segment.

The sections below further discuss the exceptions.

Invalid OP-Code (Interrupt 6)

When the execution unit spots an invalid opcode, Interrupt 6 is activated. The invalid opcode is not spotted until an attempt is made to execute it (i.e., this exception is not produced by prefetching an invalid opcode). The stored CS:IP will point to the invalid opcode or any leading prefixes, and no error code will be pushed on the stack. The exception, which can be handled within the same task, may be resumed.

This exception results in all cases of an invalid operand. An inter-segment jump referencing a register operand or an LES instruction with a register source operand are two examples.

Double Fault (Interrupt 8)

Interrupt 8 (Double Fault) occurs if two different faults happen within a single instruction, and if the first fault is #0, #10, #11, #12, or #13 (i.e., a General Protection fault in level 3 is succeeded by a Not Present fault due to a Not Present segment). The 80C286 enters shutdown if another protection violation happens during the processing of Interrupt 8. During this time, no more instructions or exceptions are processed.

The CPU can be brought out of shutdown by either NMI or RESET. If no errors result while the NMI interrupt is being processed, an NMI input can force the CPU out of shutdown. In any other situation, shutdown can only be left by the RESET input. NMI

secures the CPU in protected mode, and RESET makes it leave protected mode. A HALT bus operation with A1 Low externally signals shutdown.

A task gate must be used so the Double Fault handler can assure an appropriate task state to react to the exception. The back link field in the present TSS will define the TSS of the task creating the exception. The saved address will point to the instruction that was being performed (or was ready to perform) when the error was spotted, and the error code will be nonexistent.

The Double Fault interrupt does not result when noticing a new exception while attempting to activate handlers for the following interrupts: 1, 2, 3, 4, 5, 6, 7, 9, and 16.

Processor Extension Segment Overrun (Interrupt 9)

Interrupt 9 indicates that the processor extension (like the AMD 80C287 math coprocessor) has surpassed the limit of a segment while trying to read/write the second or successive words of an operand. The processor extension data channel within the 80C286 produces the interrupt during the limit test done on each data transfer between memory and the processor extension. Although it cannot be restarted, this interrupt can be handled in the same task.

The processor extension causes Interrupt 9, an asynchronous demand, by referencing something beyond a segment boundary, as in the case of all external interrupts. Since Interrupt 9 may be generated any time after the processor extension is started, the 80C286 does not retain any information that defines what specific operation began in the processor extension. The processor extension keeps special registers that define the last instruction it performed and the desired operand's address.

Following this interrupt, no WAIT or escape instruction, aside from FNINIT, can be performed until the interrupt condition is removed or the processor extension is reset. The interrupt indicates that the processor extension is asking for an invalid data transfer, and the extension is constantly busy when awaiting data. If the CPU performs an instruction that forces it to wait for the processor extension before resetting it, the result is deadlock. Deadlock refers to the CPU waiting for the processor extension to become idle, while the processor extension waits for the CPU to carry out its data request.

The FNINIT instruction will reset the processor extension while avoiding deadlock. This restriction is waived after the interrupt is removed. Then the instruction and operand address may be read through FSTENV or FSAVE. The result is the segment overrun in the processor extension's special registers.

The task disrupted by Interrupt 9 may not be the task that performed the ESC instruction that caused the interrupt. The operating system should trace which task used the AMD 80C287 math coprocessor last. The interrupted task can be restarted if it did not perform the ESC instruction, but the task that performed the ESC instruction cannot.

Invalid Task State Segment (Interrupt 10)

Interrupt 10 is activated if, during a task switch, the new TSS that the task gate points to is invalid. The EXT bit signals whether the exception was caused by an event beyond the program's control.

Table 9-5 lists the cases in which a TSS is considered invalid.

Once the presence of the new TSS is confirmed, the task switch is considered concluded, with the back link set to the previous task if need be. Each error is managed in the context of the new task.

Interrupt 10 must be handled via a task gate to guarantee an appropriate TSS to process it, and the handler must reset the busy bit within the new TSS.

Not Present (Interrupt 11)

If an attempt is made to load a Not Present segment or to use a control descriptor that is labeled Not Present, Interrupt 11, completely restartable, will occur. (But if the missing segment is an LDT that a task switch requires, Interrupt 10 results.)

It is possible for any segment load instruction to create this exception. Interrupt 11 is always treated in the context of the task in which it appears.

Table 9-5 indicates the error code's form. If an event external to the program created an interrupt that later referenced a Not Present segment, the EXT bit is set. If the error code references an IDT entry (that is, an INT instruction that refers to a Not Present gate), bit 1 is set. The upper 14 bits belong to the concerned segment selector.

During a task switch and while a Not Present interrupt appears, the ES and DS segment registers may not reference memory (the selector values are loaded prior to the descriptors being checked). The Not Present handler should not depend on being able to use the values located in ES, SS, and DS without producing another exception, because the actual task switch may have altered the registers' values. The exception surfaces in the new task and the return pointer indicates the first instruction of the new task.

Caution: The loading of the DS or ES descriptors may not be finished. Appropriate loading of the DS and ES descriptors should be verified by the Interrupt 11 handler before the first instruction of the new task is executed.

Table 9-5 Conditions That Invalidate the TSS

Reason	Error Code
The limit in the TSS descriptor is less than 43	TSS ID + EXT
Invalid LDT selector or LDT not present	LDT ID + EXT
Stack segment selector is null	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	SS ID + EXT
Code segment selector is outside table limit	CS ID + EXT
Code segment selector does not refer to code segment	CS ID + EXT
Non-conforming code segment DPL ≠ CPL	CS ID + EXT
Conforming code segment DPL > CPL	CS ID + EXT
DS or ES segment selector is outside table limits	ES/DS ID + EXT
DS or ES are not readable segments	ES/DS ID + EXT

Stack Fault (Interrupt 12)

Interrupt 12 results from stack underflow or overflow results as well as from a Not Present stack segment referenced in the process of an inter-task or inter-level transition. This exception is entirely restartable. An error code of 0 occurs if there is a limit violation of the present stack. The error code's EXT bit indicates if an interrupt external to the program induced the exception.

This exception can result if any instruction loads a selector to SS (POP SS, task switch). If there is a chance that any level 0 stack may not be present, a task gate must be used by this exception.

When there is a stack fault, the ES and DS segment registers may not be able to reference memory. The selector values, in the process of a task switch, are loaded prior to checking the descriptors. The stack fault handler should check the stored values of SS, CS, DS, and ES before recovering them to make sure that they reference present segments.

General Protection Fault (Interrupt 13)

If a protection violation not discussed in the above paragraphs occurs, it is labeled as Interrupt 13, a general protection fault. The error code is zero for the following: limit violations, write to read-only segment violations, and accesses relative to DS or ES when they are zero or reference a segment at a higher privilege level than CPL. Other access violations (an incorrect descriptor type) push a non-zero error code that defines the selector used on the stack. A restartable state is signaled by error codes with bit 0 cleared and bits 15–2 non-zero.

Bit 1 of the error code indicates if the selector is in the IDT or LDT/GDT. Bit 2 isolates LDT from GDT, if bit 1 = 0. Bit 0 (EXT) identifies if the program or an event external to the program, such as single stepping, an external interrupt, a processor extension Not Present, or a segment overrun, is responsible for the exception. The selector usually has nothing to do with the instruction that was disrupted, if bit 0 is set. Instead, the selector references some failed aspect of interrupt servicing.

Except for processor extension segment overrun exceptions, the disrupted program is restartable when bit 0 of the error code is set. The exception with the error code where bit 0 = 1 means that some interrupt has been lost because of a fault in the descriptor the error code indicated.

A non-zero error code with bit 0 cleared may be the interrupted instruction's operand, an operand from a gate the instruction references, or a field from the invalid TSS.

When a General Protection interrupt happens in the process of a task switch, the ES and DS segment registers may not reference memory (the selector values are loaded prior to checking the descriptors). The general protection handler should not depend on being able to use the values in ES, SS, and DS without creating another exception, because the actual task switch may have altered the registers' values. The exception appears in the new task and the return pointer indicates the first instruction of the new task.

Caution: The loading of the DS or ES descriptors may not be finished. Appropriate loading of the DS and ES descriptors should be verified by the Interrupt 13 handler before the first instruction of the new task is executed.

In real address mode, Interrupt 13 results if software tries to read or write a 16-bit word at segment offset 0FFFFH.

ADDITIONAL INTERRUPTS

The interrupts that have not been described are Interrupt 0, the divide-error exception, Interrupt 5, the bound-range exceeded exception, and Interrupt 1, the single-step interrupt. The divide-error or bound-range exceptions make it seem as though that instruction never executed. That is, the registers are restored and the instruction may be restarted. The divide-error exception appears in the process of a DIV or an IDIV instruction in which the quotient is too great to be represented or the divisor is zero.

When a value surpasses the limit set for it, Interrupt 5 results. A program can make use of the BOUND instruction to compare a signed array index against signed limits identified in a two-word memory block. The block might be located just in front of the array to ease addressing. The block's first word indicates the array's lower limit, the second word defines the array's upper limit, and a register identifies the array index to be checked.

Single Step Interrupt (Interrupt 1)

Interrupt 1 lets programs perform a single instruction at a time. This single stepping is governed by the Flag Word's TF bit. Once this bit is set, an internal single-step interrupt will appear following the next instruction's execution. The interrupt stores the flags and return address on the stack, clears the TF bit, and applies an internally supplied vector of 1 to shift control to the service routine through the IDT.

The IRET instruction or a task switch is necessary to set the TF bit and to shift control to the subsequent instruction to be single stepped. If TF=1 in a TSS and that task is activated, it will be interrupted after it performs the first instruction.

The Single Step flag is typically not cleared by privilege modifications within a task. However, TF is cleared by INT instructions. Thus, software debuggers that single-step code must acknowledge and imitate INT n or INT 0 instead of performing them directly. System software should inspect the present execution privilege level after any single-step interrupt to determine if single stepping should proceed.

The interrupt priorities in hardware ensure that single stepping ends if there is an external interrupt. The single-step interrupt is given priority, and thus the TF bit is cleared, when both an external interrupt and a single step interrupt occur simultaneously. Once the return address or switching tasks are saved, the external interrupt input is checked before the Single Step handler's first instruction performs. The external interrupt is then processed if it is still pending, and the external interrupt handler is not single stepped. Thus, simply single step an interrupt instruction referring to the interrupt handler in order to single step an interrupt handler.

CHAPTER 10

SYSTEM CONTROL AND INITIALIZATION

Special flags, registers, and instructions control the significant processes and interaction in 80C286 operations. The flag register contains 3 bits that indicate the present I/O privilege level (IOPL: 2 bits) and the nested task bit (NT). Four other registers maintain the virtual addressing and memory protection features. One register points to the present Task State Segment, while the other three point to the memory-based descriptor tables: GDT, LDT, and IDT. The following section describes these flags and registers.

SYSTEM FLAGS AND REGISTERS

The IOPL flag, or bits 12 and 13 of the flags word, directs access to I/O operations and interrupt control instructions. These two bits exemplify the highest privilege level (maximum numerical CPL) at which the task may execute I/O instructions. Changing the IOPL flags is limited to programs at level 0 or to a task switch.

IRET uses the NT flag to choose the appropriate return: the regular return in a task is executed if NT = 0. As Chapter 8 described, the Nested Task flag (bit 14 of flags) is set when a task instigates a task switch through a CALL or INT instruction. The old and new task state segments are stamped busy, while the back link field of the new TSS is set to the old TSS selector. Once the old NT state is saved, an interrupt that does not instigate a task switch will clear NT. A zero selector should be set in the back link field of the TSS to stop a program from invoking an illegal task switch by setting NT and then performing IRET. In fact, an illegal task switch that uses IRET will result in Exception 13. When flags are recovered from the stack, the instructions POPF and IRET can set or clear NT, and these same instructions can modify the interrupt enable flag, as well. If CPL ≤ IOPL, then POPF and IRET can alter the Interrupt Flag (IF). In any other case, these instructions disregard the state of the IF bit in the new flag word.

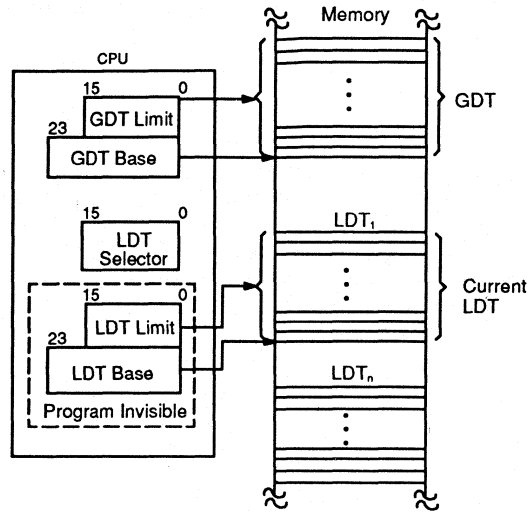
Note: The CLI and STI instructions are good only when CPL ≤ IOPL; exception 13 results if this is not the case.

Descriptor Table Registers

The three descriptor tables used for all memory accesses are based at addresses provided by (saved in) the following registers: the Global Descriptor Table Register (GDTR), the Interrupt Descriptor Table Register (IDTR), and the Local Descriptor Table Register (LDTR). Each one includes a 24-bit base field and a 16-bit limit field. The base field supplies the real memory address of the table's start, while the limit field indicates the greatest offset allowed in accessing table entries (see Figures 10-1, 10-2, and 10-3).

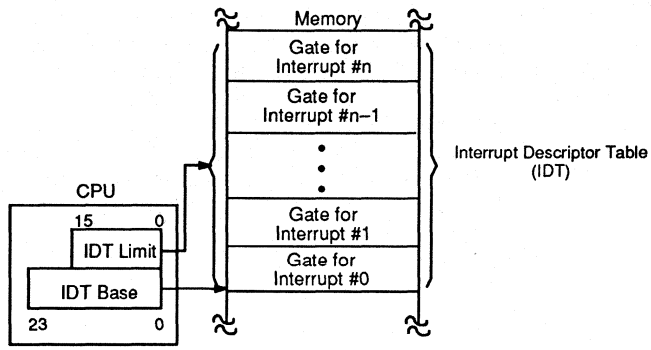
The LDTR includes a selector field that specifies the descriptor for that table, as well. LDT descriptors are required to stay in the GDT.

Figure 10-1 **Local and Global Descriptor Table Definition**



09729B-015

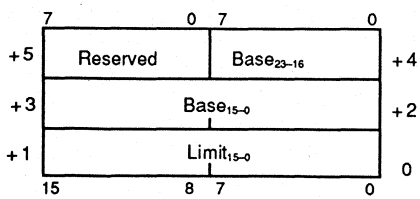
Figure 10-2 **Interrupt Descriptor Table Definition**



The IDT may contain interrupt gates, traps or task gates only.

09729B-017

Figure 10-3 **Data Type for Global Descriptor Table and Interrupt Descriptor Table**



09729B-016

The task register (TR), which points to the task state segment for the presently executing task, is comparable to a segment register, with selector, base, and limit fields. Of these fields, only the selector field is readable in normal situations. Each such

selector is a distinct definer of its task. Chapter 8 describes the task register's functions.

The following section discusses the instructions which control these special registers.

SYSTEM CONTROL INSTRUCTIONS

The instructions loading the GDTR and IDTR from memory can be performed only in real address mode or at privilege level 0; if they are not, exception 13 occurs. The store instructions for GDTR and IDTR need not be performed at a specific privilege level. LIDT, LGDT, SIDT, and SGDT are the four instructions that move 3 words between the defined descriptor table register and the effective real memory address provided (see Figure 10-3). The format of the 3 words includes a 2-byte limit and a 3-byte real base address, succeeded by an unused byte. These instructions are typically performed in the process of system initialization.

The LLDT instruction, which loads the LDT registers from a descriptor in the GDT, uses a selector operand to that descriptor instead of referencing the descriptor directly. If LLDT is not executed at privilege level 0, exception 13 will occur. LLDT is generally necessary only during system initialization due to the processor instantly exchanging the LDTR contents as part of the task-switch procedure.

The TSS or the register caches are not automatically updated when an LLDT instruction is executed. To properly alter the LDT of the presently active task so that the change is maintained across task switches, the following three steps must be executed sequentially:

1. Save the new LDT selector in the proper word of TSS.
2. Load the new LDT selector into LDTR.
3. Reload the DS and ES registers if they reference LDT-based descriptors.

Note: The present code segment and stack segment descriptors should either remain in the GDT or be duplicated to the identical location in the new LDT.

SLDT (store LDT), which stores the local descriptor table selector from the LDTR register's program visible portion, may be performed at any privilege level.

Again, Task Register loading or storing is comparable to that of the LDT. The LTR instruction, which only operates at level 0, loads the LTR at initialization time with a selector for the original TSS. LTR simply alters the present TSS; it does not cause a task switch.

Note: The old TSS descriptor's busy bit is not modified, while the new TSS selector's busy bit must be zero and will be set by LTR. The LDT and any segment registers which reference the old LDT should be reloaded. STR, which lets TR contents be saved into memory, can be performed at any privilege level. Since the TR is controlled by the task-switch operation, LTR is not normally required following initialization.

Machine Status Word

The Machine Status Word (MSW), which identifies the configuration and status of the 80C286, is not part of a task's state. The LMSW instruction performed in real address mode or at privilege level 0 only loads the MSW word, or the SMSW instruction performing at any privilege level stores it. The 80C286 uses the lower four bits of this 16-bit register. Table 10-1 lists the meanings of these four bits.

Table 10-1 **MSW Bit Functions**

Bit Position	Name	Function
0	PE	Protected mode enable places the 80C286 into protected mode and cannot be cleared except by RESET.
1	MP	Monitor processor extension allows WAIT instructions to cause a processor extension Not Present exception (number 7) if TS is also set.
2	EM	Emulate processor extension causes a processor extension Not Present exception (number 7) on ESC instructions to allow a processor extension to be emulated.
3	TS	Task switched indicates the next instruction using a processor extension will cause Exception 7, allowing software to test whether the current processor extension context belongs to the current task.

The TS flag is set under hardware control and reset under software control. Once the TS flag is set, the next instruction that uses a processor extension results in a processor extension Not Present exception (#7). This feature lets software determine if the present processor extension state belongs to the present task. The software can store the state of any processor extension with the state of the task using it, if the present processor extension state belongs to a separate task. Therefore, the TS bit shields a task from processor extension errors that are caused by an earlier task's actions.

Once the exception handler has set up the appropriate processor extension state, the CLTS instruction, which executes at privilege level 0 only, resets the TS flag.

The EM flag reveals that software is to imitate a processor extension function. If EM = 1 and MP = 0, the processor extension Not Present exception (#7) traps all ESCAPE instructions.

MP flag indicates if a processor extension is present. Exception 7 results from escape and wait instructions, if MP = 1 and TS = 1.

Either the MP or the EM bit (but not both) must be set, if ESC instructions are to be executed.

The PE flag denotes that the 80C286 is in the protected virtual address mode. Only a reset, which places the system in real address mode emulating the 8086, can clear a PE flag once it is set.

The suggested usage of the MSW is shown in Table 10-2. Additional encodings of such bits are not advisable.

Other Instructions

Instructions that confirm or modify access rights, segment limits, or privilege levels can be used to avoid exceptions or faults that can be corrected.

Table 10-2 Recommended MSW Encodings for Processor Extension Control

TS	MP	EM	Recommended Use	Instructions Causing Exception 7
0	0	0	Initial encoding after RESET. 80C286 operation is identical to 8086, 8088. Use this encoding only if no ESC instructions are to be executed.	None
0	0	1	No processor extension is available. Software will emulate its function. Wait instructions do not cause Exception 7.	ESC
1	0	1	No processor extension is available. Software will emulate its function. The current processor extension context may belong to another task.	ESC
0	1	0	A processor extension exists.	WAIT (if TS = 1)
1	1	0	A processor extension exists. The current processor extension context may belong to another task. The Exception 7 on WAIT allows software to test for an error pending from a previous processor extension operation.	ESC or WAIT (if TS = 1)

PRIVILEGED AND TRUSTED INSTRUCTIONS

Instructions that perform only when CPL = 0 are "privileged". A General Protection exception (#13) with an error code of zero occurs if an attempt is made to perform the "privileged" instructions at another "privileged" level. The "privileged" instructions control descriptor tables or system registers. An unrecoverable state is the result of invalid use of these instructions. Some of these (LGDT, LLDT, and LTR) were discussed earlier in this chapter.

Additional "privileged" instructions include:

- LIDT—Load interrupt descriptor table register
- LMSW—Load machine status word
- CLTS—Clear task switch flag
- HALT—Halt processor execution
- POPF (POP flags) or IRET can alter the IF value only if the user is working at a "trusted privilege" level. POPF only alters IOPL at Level 0.

"Trusted" instructions are limited to performance at a "privilege" level of CPL ≥ IOPL. For every task, the operating system specifies a privilege level below which these instructions are unusable. The majority of these instructions concern input/output or interrupt management. The IOPL field in the flag word that contains the privilege level restriction can be altered only when CPL = 0. The "trusted" instructions are listed below:

- Input/Output—Block I/O, Input, and Output: IN, INW, OUT, OUTW, INSB, INSW, OUTSB, OUTSW
- Interrupts—Enable Interrupts, Disable Interrupts: STI, CLI
- Other—Lock Prefix

INITIALIZATION

Specific registers are set to values already defined whenever the 80C286 is initialized or reset. User software must perform all other desired initialization. (Appendix A has an example of a 286 initialization routine.) RESET causes the 80C286 to end all execution and local bus activity. While RESET is active, instruction or bus action will not occur. In real address mode, execution starts after RESET is no longer active and an internal processing interval (3–4 clocks) results. Below is the initial state at reset:

```
FLAGS = 0002
MSW = FFF0H
IP = FFF0H
CS.Selector = F000H   CS.base = FF0000H     CS.limit = FFFFH
DS.Selector = 0000H   DS.base = 000000H     DS.limit = FFFFH
ES.Selector = 0000H   ES.base = 000000H     ES.limit = FFFFH
IDT.base = 000000H    IDT.limit = 03FFH
```

The system initialization area and the interrupt table area, both fixed areas of memory, are reserved. The system initialization area starts at FFFFF0H (through FFFFFFH). The interrupt table area, which is not reserved, starts at 000000H (through 0003FFH).

Segment registers are correct and protection bits are set to 0 by this point. With PE = 0, the 80C286 starts operation in real address mode, and maskable interrupts are disabled. No processor extension is presumed or emulated (EM = MP = 0).

In order to permit access to the first 64K of memory (just as in the 8086), DS, ES, and SS are initialized at reset. The CS:IP combination indicates a beginning address of FFFF0H. The four most significant bits are not used in real address mode. Thus, the same FFF0H address as the 8086 reset location is supplied. A bootstrap loader at the high end of the address space can permit use of (or upgrade to) the protected mode. Location FFF0H usually includes a JMP instruction whose focus is the actual start of a system initialization or restart program.

Following RESET, CS points to the uppermost 64K bytes of the 16-Mb physical address space. Reloading CS register via control transfer to a separate code segment in real address mode will set zeros in the top 4 bits. All of the top 64K bytes of address space may be used for initialization because the original IP is FFF0H.

Real Address Mode

1. Allocate a stack.
2. Load programs and data into memory from secondary storage.
3. Initialize external devices and the Interrupt Vector Table.
4. Set registers and MSW bits to preferred values.
5. After providing that there is a correct interrupt handler for each possible interrupt, set FLAG bits to preferred values, and the IF bit to enable interrupts.
6. Execute (normally through an inter-segment JMP to the primary system program).

Protected Mode

Extra steps needed to operate the complete 80C286 virtual address mode initialization process correctly are listed below:

1. Load programs and related descriptor tables.
2. Load correct GDT and IDT descriptor tables, setting the GDTR and IDTR to their proper value.
3. Set the PE bit to enter protected mode.
4. To clear the processor queues, perform an intra-segment JMP.
5. Load or create a proper task state segment for the first task to be performed in protected mode.
6. If an LDT is not necessary, load the LDTR selector from the task's GDT or 0000H (null).
7. Set the stack pointer (SS, SP) to a proper location in a correct stack segment.
8. Stamp all items that do not appear in memory as not-present.
9. Set FLAGS and MSW bits to proper values for the preferred system configuration.
10. Initialize external devices.
11. Make sure that each possible interrupt has a proper interrupt handler.
12. Enable interrupts.
13. Execute.

The steps needed to load every required table and register that allows execution of the protected mode system's initial task is shown in Appendix A. The program in Appendix A presumes that development tools have been utilized to produce a prototype GDT, IDT, LDT, TSS, and all the data segments required to activate that initial task. Normally, these items are stored on EPROM, but they must all be duplicated into RAM to start on most systems, otherwise, the 80C286 will attempt to set the accessed or busy bits by writing into the EPROM.

The builder, by using appropriate naming conventions, can allocate alias data segments that are bigger than the prototype EPROM version. The example's code will zero out the additional entries to promote subsequent dynamic usage.

CHAPTER 11

ADVANCED TOPICS

Advanced topics, such as virtual memory management, restartable instructions, special segment attributes, and the validation of descriptors and pointers, are the subject of this chapter.

VIRTUAL MEMORY MANAGEMENT

The Not Present fault (Exception 11, or 12 for stacks) results whenever access to a segment is desired and the access byte in its descriptor discloses that the segment is not present in real memory. This fault's handler can be set up to bring the missing segment to real memory (transferring or overwriting another segment if needed), or to stop performance of the requesting program if this is not feasible.

The accessed bit (bit 0) of the access byte is furnished in both executable and data segment descriptors to maintain profiling of segment usage. Whenever the 80C286 hardware accesses the descriptor, the A-bit is set in memory. This is applicable to selector test instructions (discussed below) and to segment register loading. Reading the access byte and restoring it with the A-bit set is an indivisible operation. That is, it is executed as a read-modify-write with bus lock. An operating system can distinguish segments of low or zero access and select among them for replacement, provided it formulates a segment usage profile over time.

The task that desired access to a Not Present segment can continue its execution when the segment is brought into real memory since all instructions that load a segment register can be restarted.

Not Present exceptions can only occur on segment register load operations, gate accesses, and task switches. The stored instruction pointer references the initial byte of the violating instruction. All other facets of the saved machine state remain just as they were prior to executing the violating instruction. The program continues performing after the fault handler settles the fault condition and executes an IRET. The only external hint of a segment trade is the extra performance time.

SPECIAL SEGMENT ATTRIBUTES

Conforming Code Segments

Code segments meant to be used at privilege levels that may be different require a feature that lets them emulate the privilege level of the calling task. Such segments, called conforming segments, are also helpful for interrupt-driven error routines that must only be as privileged as the routine causing the error.

Bit 2 of a conforming code segment's access byte is set to 1. In other words, a CALL or JMP instruction in a task of the same or lower privilege (i.e., the task's CPL is numerically greater than or equal to this segment's DPL) can refer to it. When executing the conforming code segment, CPL stays the same, and a conforming segment still uses the stack from the CPL. This is the one instance where a code segment's DPL can be numerically less than the CPL. The segment is not conforming and only a task of CPL = DPL can reference it if bit 2 is 0.

Inter-segment returns that reference conforming code segments use the RPL field of the return address' code selector to specify the new CPL. When the conforming code segment DPL≤RPL, the RPL becomes the new CPL.

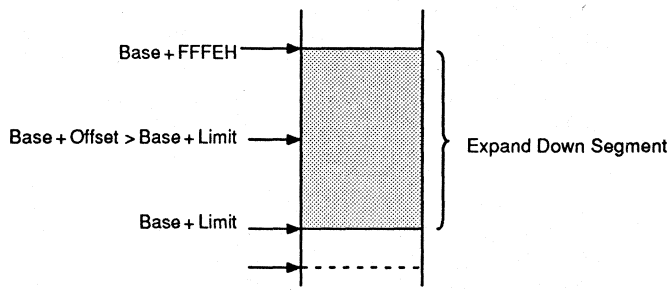
A conforming segment, if readable, can be read from any privilege level without limitation, and thus is the one exception to the protection rules. The result is that constants can be saved with conforming code. A read-only look-up table, for example, can be planted in a conforming code segment that can change system-wide logical ID's into character strings that represent those logical entities.

Expand Down Data Segments

A segment is an Expand Down segment if bit 2 in its access byte is 1. Each offset that references an expand-down segment must be greater than the segment limit, contrary to regular data segments (bit 2 = 0), in which each offset must be less than or equal to the segment limit. An expand-down segment is illustrated in Figure 11-1.

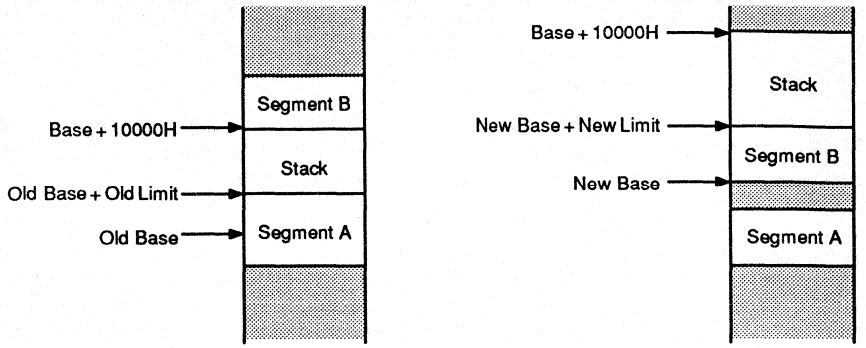
The Expand Down segment's size can be changed by altering either the base or the limit. An expand-down segment with limit = 0 has a size of $2^{16}-1$ bytes. An expand-down segment with a limit value of FFFFH is 0 bytes. The base + offset value in such a segment should always be more than the base + limit value. Thus, only by utilizing an expand-up segment can a full size segment (2^{16} bytes) be acquired.

Figure 11-1 **Expand-Down Segment**



When a protection fault discloses that a data segment's limit has been reached, the operating system should inspect the Expand-Down bit. The operating system should expand the segment limit if the Expand-Down bit is not set. If it is set, then the limit should be reduced. In either case, this provides extra room, presuming the segment is not write-protected (bit 1 is not 0). If in some instances, the operating system can determine that there is not sufficient room to increase the data segment so the need that created the fault can be met, it can transfer the data segment to a region of memory with enough space (see Figure 11-2).

Figure 11-2 Dynamic Segment Relocation and Expansion of Segment Limit



14554A-048

POINTER VALIDATION

Pointer validation, a significant part of finding programming errors, is needed to maintain separation between the privilege levels. The steps below are included in pointer validation:

1. Determine if the supplier of the pointer may access the segment.
2. Determine if the segment type is suitable for its intended use.
3. Determine if the pointer breeches the segment limit.

The 80C286 hardware automatically performs checks 2 and 3 as instruction is being executed. The software, however, must collaborate in performing the first check. Software can directly carry out steps 2 and 3 to check for possible violations (instead of invoking an exception). The unprivileged instructions (LSL, LAR, VERR, and VERW) are supplied for this intent.

The load access rights (LAR) instruction acquires the access rights byte of a descriptor that the selector used in the instruction indicates. If that selector can be seen at the CPL, the instruction loads the access byte into the identified destination register as the higher byte (the low byte being zero), and sets the zero flag. The access bits can be checked as soon as they are loaded. System segments that cannot be read or changed include a task state segment and a descriptor table. This instruction confirms that a pointer references a segment of the appropriate privilege level and type. No access value is recovered, and the zero flag is cleared, if the RPL or CPL is more than DPL or the selector is beyond the table limit. Any RPL or CPL can access conforming code segments.

Additional parameter testing can be done through the load segment limit (LSL) instruction. If the descriptor indicated by the selector (in memory or a register) is seen at the CPL, LSL loads the indicated register with a word consisting of that descriptor's limit field. This is only possible for segments, task state segments, and local descriptor tables (words from control descriptors cannot be accessed). It is the segment type's function to interpret the limit. Downward expandable data segments, for instance, handle the limit differently from the way code segments do.

If the loading was executed, the zero flag (ZF) is set for both LAR and LSL. If not, the zero flag is cleared. In real address mode, neither instruction is defined and an invalid opcode exception (Interrupt #6) occurs.

Descriptor Validation

The VERR and VERW instructions of the 80C286 determine if a selector points to a segment that can be read or written at the present privilege level. If the outcome is negative, neither instruction results in a protection fault.

VERR validates a segment for reading and loads ZF with 1 if that segment can be read from the present privilege level. The validation process determines if: 1) the selector indicates a descriptor inside the bounds of the GDT or LDT, 2) it specifies a segment descriptor (versus a control descriptor), and 3) the segment can be read and is at the proper privilege level. The privilege check for data segments and non-conforming code segments requires that the DPL be numerically greater than or equal to both the CPL and the selector's RPL. No privilege check exists for conforming segments.

VERW supplies the same capacity as VERR to verify writability. VERW, like VERR, loads ZF if the writability check's outcome is positive. The instruction determines if the descriptor is inside bounds, is a segment descriptor, is writable, and if its DPL is numerically greater than or the same as both the CPL and the selector's RPL. Code segments, whether they are conforming or not, are not writable.

Pointer Integrity: RPL and the "Trojan Horse Problem"

The Requested Privilege Level (RPL) feature can hinder improper use of pointers that could corrupt the operation of higher privilege code or data from a less privileged level.

A typical example is a file system procedure, FREAD (**file_id, nybytes, buffer-ptr**). This theoretical procedure reads data from a file into a buffer, overwriting whatever is there. Usually, FREAD would be accessible at the user level and provide only pointers to the file system procedures and data found and functioning at a privileged level. Such a procedure usually hinders user-level procedures from directly altering the file tables. Nevertheless, without typical formality for checking pointer validity, a user-level procedure could provide a pointer into the file tables, replacing its buffer pointer and making the FREAD procedure corrupt them inadvertently.

Such problems can be avoided by using the RPL field, which lets a privilege attribute be allocated to a selector. This privilege attribute would usually denote the privilege level of the code that produced the selector. The 80C286 hardware will automatically inspect the RPL of any selector loaded into a segment register or a control register to determine if the RPL permits access.

To protect against erroneous pointers, the called procedure simply needs to guarantee that all selectors transferred to it have an RPL at least as large (numerically) as the initial caller's CPL. This suggests that the selectors were no more trusted than their supplier. A protection fault will occur when loaded into a segment or control register if a selector is used to access a segment that the caller could not access directly (i.e., the RPL is numerically higher than the DPL).

The caller's CPL is accessible in the CS selector, which was pushed on the stack as the return address. A special instruction, ARPL (Adjust RPL field of selector instruction), can be used to properly modify the pointer's RPL field to become the greater of

its initial value and the value of the RPL field in a defined register. The latter is usually loaded from the caller's CS register, which is located on the stack. If the selector's RPL changes because of the adjustment, ZF is set. In any other case, the zero flag is cleared.

AMD 80C287 MATH COPROCESSOR CONTEXT SWITCHING

The task switch operation does not alter the context of a coprocessor (like the AMD 80C287 math coprocessor). A processor extension context must be modified only when a new task tries to use the processor extension, which still includes the previous task's context. The 80C286 spots the first use of a processor extension following a task switch, when the processor extension Not Present exception (#7) occurs if the TS bit is set. Then the interrupt handler may determine if a context change is needed.

The 80C286 services numeric errors only when it performs wait or escape instructions, since the processor extension is operating independently. Consequently, the numerics error from one task may not be recorded until the 80C286 starts a new task. If the 80C286 task has changed, it is best to postpone servicing that error until the initial task is recovered. For example, interrupt handlers using the AMD 80C287 math coprocessor should not have their timing upset by a numeric error interrupt that concerns some previous process. Servicing someone else's error makes little sense.

The processor-extension Not Present exception occurs (#7) if the task switch bit is set (bit 3 of MSW) when the CPU starts to perform a wait or escape instruction. This interrupt's handler must know who presently "owns" the AMD 80C287 math coprocessor. That is, the handler must know the most recent task to send a command to the AMD 80C287 math coprocessor. If the owner of that task and the present task are the same, then it was simply interrupted and the interrupt handler has since returned. Interrupt 7's handler merely clears the TS bit, recovers the functioning registers, and returns (restoring interrupts if enabled).

However, the handler must first store the existing AMD 80C287 context in the old task's save area, if the recorded owner and the present task are different. It can then rebuild the valid AMD 80C287 context from the present task's save area.

The code example in Figure 11-3 depends on the understanding that each TSS entry in the GDT is succeeded by an alias entry for a data segment that points to the identical physical region of memory that includes the TSS. The following are also included in the alias segment: an area for saving the AMD 80C287 context, the kernel stack, and certain kernel data. The initial 44 bytes in that segment are the 80C286 context, then 94 bytes for the processor extension context, sometimes followed by the kernel stack and kernel private data areas.

The understood standard is that the stack segment selector points to this data segment alias so that all of the information discussed above is directly addressable whenever an interrupt occurs at level zero and SS is loaded automatically.

It is presumed that the program example recognizes just one data segment that points to a global data area where it can locate the one word AMD 80C287 owner to start the processing explained. The particular operations needed are indicated in Table 11-1, as shown in the figure.

Figure 11-3 Example of AMD 80C287 Math Coprocessor Context Switching

```
ASSEMBLER INVOKED BY:  ASM286,86 :F5:SWNPX.A86

LOC  OBJ                    LINE     SOURCE
                               1 *1  $title('Switch the NPX Context on First Use After a Task Switch')
                               2
                               3                             name    switch_npx_context
                               4
                               5                             public  switch_NPX_context
                               6                             extrn   last_npx_task:word
                               7     ;
                               8     ;         This interrupt handler will switch the NPX context if a new task
                               9     ;       is attempting to use the NPX context of another task after a task
                              10     ;       switch.  If the NPX context belongs to the current task, nothing happens.
                              11     ;
                              12     ;       A trap gate should be placed in IDT entry 7 referring to this routine.
                              13     ;       The DPL of the gate should be 0 to prevent spoofing.  The code segment
                              14     ;       must be at privilege level 0.
                              15     ;
                              16     ;       The kernel stack is assumed to overlay the TSS and the NPX save area
                              17     ;       is placed at the end of the TSS area.
                              18     ;
                              19     ;       A global word variable LAST_NPX_TASK identifies the TSS selector of
                              20     ;       the last task to use the NPX.
                              21     ;
  002C                        22     npx_save_area           equ     word ptr 44     ; Offset of NPX save area in TSS
                              23 *1  $eject
----                          24     kernel_code             segment er public
                              25
0000                          26     switch_npx_context      proc    far wc(0)
                              27
0000 50                       28             push    ax                      ; Save working registers
0001 1E                       29             push    ds
0002 B8----           E       30             mov     ax,seg last_npx_task    ; Get address of id of last NPX task
0005 8ED8                     31             mov     ds,ax
0007 0F00C8                   32             str     ax                      ; Get id of this task
000A 24FC                     33             and     al,not 3                ; Remove RPL field
000C 0F06                     34             clts                            ; Clear task switched flag
000E FA                       35             cli                             ; No interrupts allowed!
                              36     ;
                              37     ;       Last_npx_word cannot change due to other interrupts after this point.
                              38     ;
000F 3B060000         E       39             cmp     ax,ds:last_npx_task     ; See if same task
0013 7412                     40             je      same_task
                              41
0015 87060000         E       42             xchg    ax,ds:last_npx_task     ; Set new task id and get old one
0019 050800                   43             add     ax,8                    ; Go to TSS alias
001C 8ED8                     44             mov     ds,ax                   ; Address TSS of previous NPX task
001E DD3626C00                45             fsave   ds:npx_save_area        ; Save old NPX state
0022 36DD2626C00              46             frstor  ss:npx_save_area        ; Get current NPX state
0027                          47     same_task:
0027 1F                       48             pop     ds                      ; Return to interrupted program
0028 58                       49             pop     ax
0029 CF                       50             iret
                              51
                              52     switch_npx_context      endp
                              53
----                          54     kernel_code             ends
*** WARNING #160, LINE #54, SEGMENT CONTAINS PRIVILEGED INSTRUCTIONS
                              55                             end
```

Table 11-1 **AMD 80C287 Math Coprocessor Context Switching**

Step	Operation	Lines (Figure 11-3)
1.	Save the working registers	28, 29
2.	Set up address for kernel work area	30, 31
3.	Get current task ID from Task Register	32
4.	Clear Task Switch flag to allow AMD 80C287 work	34
5.	Inhibit interrupts	35
6.	Compare owner with current task ID	37
If same owner:		
7a.	Restore working registers	48, 49
7b.	and return	50
If owner is not current task:		
8a.	Use owner ID to save old context in its TSS	42, 43, 44
8b.	Restore context of current task;	45
	restore working registers;	46
	and return	52

MULTIPROCESSOR CONSIDERATIONS

As Chapter 8 pointed out, a bus lock is applied while the task busy bit is being tested and set, to make sure that two processors do not activate the same task concurrently. Protection traps and contrary use of dynamically different segments or descriptors, however, must be addressed by an interprocessor synchronization protocol, which can use the indivisible semaphore operation of the base instruction set. Whenever several processors are operating together, coordination of interrupt and trap vectoring must be addressed as well.

No interleaving occurs on those cycles since the interrupt bus cycles are locked. A reference to a descriptor may not be changed while being obtained because descriptor caching is locked.

When a program alters a descriptor that other processors share, it should inform them of this fact with an inter-processor interrupt. The handler for this interrupt must guarantee that segment registers LDTR and TR are reloaded. If a task switch services the interrupt, this occurs automatically.

Alteration of descriptors of shared segments in multi-processor systems may necessitate updating of on-chip descriptors as well. One processor, for example, may try to stamp a shared segment's descriptor as not present while another processor is using it. Software must guarantee that the new information updates the descriptors in the segment register caches by employing a re-entrant procedure activated by an interprocessor interrupt. The handler must guarantee that segment registers LDTR and the TR are reloaded. If a task switch services the interrupt, this occurs automatically.

SHUTDOWN

A severe error condition that deters further processing can cause shutdown. The 80C286 does not execute instructions during shutdown, which is very similar to HLT in this regard. The 80C286 externally indicates shutdown as a Halt bus cycle with A1 = 0, and the NMI or RESET input will bring the 80C286 out of shutdown. Also, during shutdown, the INTR input is disregarded.

CHAPTER 12

OVERVIEW OF NUMERIC PROCESSING

INTRODUCTION TO THE AMD 80C287 MATH COPROCESSOR

The AMD 80C287 math coprocessor is a high performance arithmetic processor that expands the 80286 instruction set with floating point instructions including transcendentals, integer, and BCD conversions. The floating point operators comply with the IEEE Standard 754. When coupled with an 80286 microprocessor, the AMD 80C287 math coprocessor provides a complete solution for high performance numeric processing applications.

Today over 180 software applications take advantage of the AMD 80C287 math coprocessor. These include spreadsheets, databases, CAD, integrated software, statistics, business graphics, financial analysis, programming tools, etc. Some calculations perform up to 800% faster when using the AMD 80C287 device.

Ease Of Use

The AMD 80C287 math coprocessor provides more than just execution speed for tasks that require intensive computation. The AMD 80C287 math coprocessor offers the functionality and strength of precise numeric computation to the user.

The AMD 80C287 math coprocessor is intended to deliver precise results when programmed by using simple pencil-and-paper algorithms. The IEEE 754 standard acknowledges the fundamental need to make numeric computations both simple and safe to use.

The majority of computers, for example, can overflow when two single-precision floating-point numbers are multiplied and then divided by a third, even if the end result is a valid 32-bit number. The AMD 80C287 math coprocessor supplies the properly rounded result. Other routine examples of undesirable machine behavior in simple calculations result when determining the roots of a quadratic equation:

$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

or determining financial rate of return, which concerns the following expression: $(1+i)^n$. Simple algorithms on the majority of machines will not supply invariably accurate results or indicate when the results are inaccurate. Sophisticated numerical techniques, unknown to most programmers, are needed to get accurate results on traditional machines in all instances. Standard application programmers using simple algorithms will create much more dependable programs using the AMD 80C287 math coprocessor. Thus, the software investment necessary to produce safe, correct computation-based products is substantially reduced.

The AMD 80C287 math coprocessor has built-in facilities for commercial computing that surpass the usual numerics support for scientific applications. The AMD 80C287 device processes decimal numbers of up to 18 digits without round-off errors, executing *exact arithmetic* on integers as large as 2^{64} or 10^{18}. Rounding errors in

accounting applications may cause monetary losses that cannot be reconciled, and thus, exact arithmetic is imperative in such applications.

Sophisticated users can activate several optional, advanced facilities of the AMD 80C287 math coprocessor, including two models of infinity, directed rounding, gradual underflow, and either automatic or programmed exception-handling facilities.

These automatic exception-handling features, which allow a great deal of flexibility in numeric processing software, do not burden the programmer. While executing numeric calculations, the AMD 80C287 math coprocessor automatically spots exception conditions that may possibly ruin a calculation. On-chip exception handlers may be activated by default to handle these exceptions, produce a reasonable result, and continue performance without program interruption. On the other hand, whenever certain exception types are spotted, the AMD 80C287 math coprocessor can signal the CPU and activate a software exception handler.

Applications

The flexibility and performance of the AMD 80C287 math coprocessor make it suitable for a wide variety of numeric applications. Usually, applications that show any of the characteristics listed below can be improved by implementing numeric processing on the AMD 80C287 device:

- Numeric data vary over a broad array of values, or contain non-integral values.
- Algorithms result in extremely large or small intermediate results.
- Computations must be exact; a large number of significant digits must be used.
- Performance requirements surpass traditional microprocessors' capacity.
- Using a programming staff that is not adept in numerical techniques, invariably safe, dependable results must be delivered.

Note: The AMD 80C287 math coprocessor can not only increase the performance of systems that use real numbers and operate on multiprecision binary or decimal integer values, but it can also decrease software development costs.

Below are a few examples that illustrate how the AMD 80C287 math coprocessor might be used in specific numerics applications. These kinds of systems have been implemented previously with minicomputers in several instances. The AMD 80C287 math coprocessor lets these applications benefit from the size and cost savings of microprocessor technology for the first time.

- **Business data processing**—The capability of the AMD 80C287 math coprocessor to recognize decimal operands and generate *precise* decimal results of up to 18 digits highly simplifies accounting programming. Financial calculations using power functions can benefit from the AMD 80C287 exponentiation and logarithmic instructions. Also, common spreadsheet and database applications can take advantage of the AMD 80C287 math coprocessor to speed-up operations.
- **Process control**—The AMD 80C287 math coprocessor, which responds to dynamic range problems automatically permits fine-tuning of control functions for more correct and effective performance. While the AMD 80C287 device's speed can be taken advantage of in real-time operations, the AMD 80C287 math coprocessor control algorithms also lend to advanced dependability and safety.

- **Computer numerical control (CNC)**—The AMD 80C287 math coprocessor can shift and place machine tool heads accurately in real-time. The hardware trigonometric support supplied by the AMD 80C287 math coprocessor improves axis positioning, as well.
- **Robotics**—Coupling small size and light power requirements with significant computational abilities, the AMD 80C287 math coprocessor is perfect for on-board six-axis positioning.
- **Navigation**—With the AMD 80C287 math coprocessor, extremely small, lightweight, and precise inertial guidance systems can be implemented. The AMD 80C287 math coprocessor, with its built-in trigonometric functions, can simplify the calculation of position from bearing data, and generate it faster.
- **Graphics terminals**—The AMD 80C287 math coprocessor can be used in graphics terminals to locally execute many functions that would, under normal conditions, require the attention of a main computer. Rotation, scaling, and interpolation are among these functions.
- **Data acquisition**—The AMD 80C287 math coprocessor can scan, scale, and reduce large amounts of data as it is collected, reducing the amount of storage and time needed to process the data for analysis.

The examples mentioned above illustrate traditional numerics applications. Additionally, there are several other types of systems that the end user does not regard as computational, but can benefit from the AMD 80C287 math coprocessor. In fact, the AMD 80C287 device provides the imaginative system designer with an opportunity as great as the introduction of the microprocessor itself. The majority of applications can be regarded as numerically-based, but only if adequate computational power is available to uphold this view. This is similar to the thousands of profitable products that have been built around buried microprocessors, although the products themselves barely resemble computers.

Upgradability

The 80C286 CPU's architecture is particularly adapted to use an AMD 80C287 math coprocessor, just by plugging in the device. Because of this, designers of 80C286 systems may want to include the AMD 80C287 math coprocessor in their designs in order to provide two levels of price and performance at little extra cost.

Two capabilities of the 80C286 CPU make the design and support of upgradable 80C286 systems especially easy:

- The 80C286 can be programmed to acknowledge the presence of an AMD 80C287 math coprocessor. In other words, software can determine whether it is operating with a AMD 80C287 math coprocessor.
- Once the 80C286 CPU has determined whether the AMD 80C287 math coprocessor is available, it can be instructed to allow the AMD 80C287 math coprocessor to perform all numeric instructions. The 80C286 CPU can emulate all AMD 80C287 numeric instructions in software, if an AMD 80C287 math coprocessor is unavailable, and this emulation is completely transparent to the application software. (The 80C286 and AMD 80C287 systems may use the same object code.) Relinking or recompiling application software is not required; the same code will simply perform quicker with an AMD 80C287 math coprocessor.

To facilitate upgradable 80C286 systems, there are software emulators available that provide the functional equivalent of the AMD 80C287 hardware, which was implemented in software on the 80C286. The operation of this emulator is identical, with the exception of timing. When the emulator becomes part of the systems software, the system with emulation and the system with hardware are effectively indistinguishable to an application program. This makes it simple for software developers to keep one set of programs for both systems. Without requiring any changes in the user's software, system manufacturers can promote the AMD 80C287 math coprocessor as an easy plug-in performance option.

PROGRAMMING INTERFACE

The 80286/AMD 80C287 configuration is programmed as one processor; a programmer sees all of the AMD 80C287 math coprocessor registers as extensions of the basic 80286 register set. ESCAPE instructions, one class of 80C286 instructions, all have a common format, and are numeric instructions. These numeric instructions, along with 80C286 instructions, are encoded into the instruction stream.

When programming the AMD 80C287 math coprocessor, any CPU memory-addressing modes may be used, allowing easy access to record structures, numeric arrays, and other memory-based data structures. The CPU's memory management and protection features are also extended to the AMD 80C287 math coprocessor.

Numeric processing is organized around the AMD 80C287 math coprocessor register stack. Programmers can regard these eight 80-bit registers as one of two things: either a fixed register set, with instructions operating on clearly-designated registers, or a classical stack, with instructions operating on the highest one or two stack elements.

The AMD 80C287 math coprocessor retains all numbers internally in a uniform 80-bit temporary-real format. Operands that may appear in memory as 16-, 32-, or 64-bit integers, 32-, 64-, or 80-bit floating-point numbers, or 18-digit packed BCD numbers, are automatically changed into temporary-real format when loaded into the AMD 80C287 registers. Computation results are later changed back into one of these destination data formats when they are sent from registers to memory.

The AMD 80C287 math coprocessor supports the seven data types listed in Table 12-1, which shows the data format for every type. All operands are retained in memory with the least significant digits beginning at the initial (lowest) memory address. With this one initial address, numeric instructions can access and store memory operands. All operands should begin at even memory addresses for optimal system performance.

Table 12-1 Numeric Data Types

Data Type	Bits	Significant Digits (Decimal)	Approximate Range (Decimal)
Word integer	16	4	$-32{,}768 \le x \le +32{,}767$
Short integer	32	9	$-2\times10^{9} \le x \le +2\times10^{9}$
Long integer	64	18	$-9\times10^{18} \le x \le +9\times10^{18}$
Packed decimal	80	18	$-99...99 \le x \le +99...99$ (18 digits)
Short real	32	6–7	$8.43\times10^{-37} \le \lvert x\rvert \le 3.37\times10^{38}$
Long real	64	15–16	$4.19\times10^{-307} \le \lvert x\rvert \le 1.67\times10^{308}$
Temporary real	80	19	$3.4\times10^{-4932} \le \lvert x\rvert \le 1.2\times10^{4932}$

The AMD 80C287 math coprocessor instructions are listed by class in Table 12-2. Since all instructions and data types are directly supported by applicable assemblers and high-level languages, special programming tools are not needed.

In fact, many development tools which support the 8086 and 8087 will support the 80C286 and AMD 80C287 math coprocessor operating in real address mode as well.

Without demanding an understanding of the 80C286 and AMD 80C287 math coprocessor chip architecture, high-level languages grant programmers access to the computational power and speed of the AMD 80C287 math coprocessor. These high-level languages handle concurrency and data synchronization, two architectural considerations, automatically. A subsequent section of this supplement describes particular rules for the assembly language programmer to handle these issues.

Table 12-2 Principle AMD 80C287 Math Coprocessor Instructions

Class	Instruction Types
Data Transfer	Load (all data types), Store (all data types), Exchange
Arithmetic	Add, Subtract, Multiply, Divide, Subtract Reversed, Divide Reversed, Square Root, Scale, Remainder, Integer Part, Change Sign, Absolute Value, Extract
Comparison	Compare, Examine, Test
Transcendental	Tangent, Arctangent, 2^x-1, $y\cdot\log_2(x+1)$, $y\cdot\log_2(x)$
Constants	0, 1, π, $\log_{10}2$, $\log_e2$, $\log_210$, $\log_2e$
Processor Control	Load Control Word, Store Control Word, Store Status Word, Load Environment, Store Environment, Save, Restore, Clear Exceptions, Initialize, Set Protected Mode

Hardware Interface

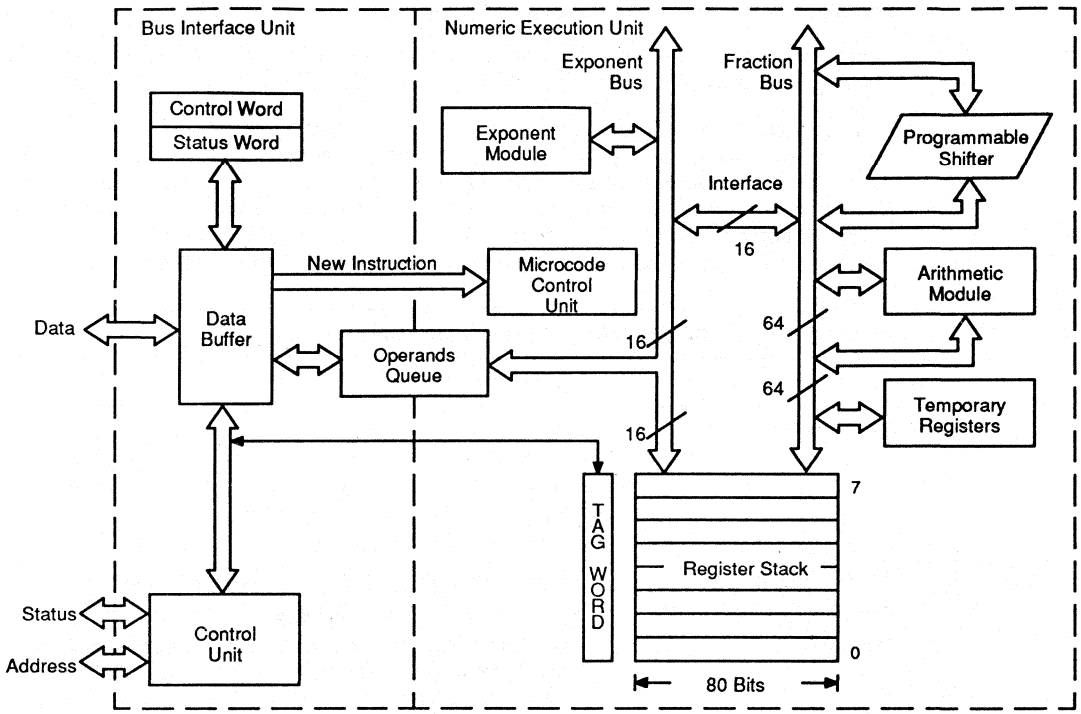
The AMD 80C287 math coprocessor is wired quite similarly to the 80C286 CPU. Four special status signals (PEREQ, $\overline{\text{PEACK}}$, $\overline{\text{BUSY}}$, and $\overline{\text{ERROR}}$) let the two processors coordinate their activities.

Figure 12-1 shows the Bus Interface Unit (BIU) and the Numeric Execution Unit (NEU), the two processing elements that the AMD 80C287 math coprocessor is split into internally. Of these two units, which operate independently of each other, the BIU can receive and decode instructions, request operand transfers with memory, and perform processor control instructions; the NEU can process individual numeric instructions.

The BIU controls the status and signal lines linking the AMD 80C287 math coprocessor and the 80C286. The NEU performs all instructions which concern the register stack, such as arithmetic, logical, transcendental, constant, and data transfer instructions. The NEU data path is 84 bits wide (68 fraction bits, 15 exponent bits, and a sign bit), and thus, internal operand transfers can be executed at extremely high speeds.

The AMD 80C287 math coprocessor performs one numeric instruction at a time. Before performing the majority of ESC instructions, the 80C286 checks the $\overline{\text{BUSY}}$ pin. Prior to initiating the command, the 80C286 waits until the AMD 80C287 math coprocessor is free, but once initiated, the 80C286 proceeds with program execution,

Figure 12-1 AMD 80C287 Math Coprocessor Block Diagram



and the AMD 80C287 math coprocessor performs the numeric instruction. These WAIT instructions differ from the 8087, which needed a WAIT instruction to check the BUSY signal before each ESC opcode, in that they are allowed, but not required, in AMD 80C287 math coprocessor programs.

In all instances, a WAIT or ESC instruction should be included after any AMD 80C287 math coprocessor store to memory (with the exception of FSTSW or FSTCW) or load from memory (with the exception of FLDENV, FLDCW, or FRSTOR) prior to the 80C286 reading or changing the memory value.

All data transfers between memory and the AMD 80C287 math coprocessor are executed when necessary by the 80C286 CPU, through its Processor Extension Data Channel. The timing of the numeric data transfers that the 80C286 executes is the same as any other bus cycle. The 80C286 memory management and protection mechanisms monitor all such transfers.

The AMD 80C287 math coprocessor can simply be thought of as an extension of the 80C286 processor from the programmer's view. The 80C286 automatically controls all interaction, which is transparent to the software, between the 80C286 and the AMD 80C287 processors on the hardware level.

The 80C286 accesses the reserved I/O port addresses 00F8H, 00FAH, and 00FCH (I/O ports numbered 00F8H through 00FFH are reserved for the 80C286/AMD 80C287 interface) to interface the AMD 80C287 math coprocessor. These I/O

operations, executed automatically by the 80C286, are separate from I/O operations resulting from program I/O instructions. I/O operations that result from the performance of ESC instructions are fully transparent to software. Any program, regardless of its current I/O Privilege Level (IOPL), can perform ESCAPE (numeric) instructions.

Programs must not execute any direct I/O operations to any of the eight ports reserved for the AMD 80C287 math coprocessor to assure its accurate operation. The IOPL of the 80C286 can shield the integrity of AMD 80C287 computations in multiuser applications, and thus, prevent any kind of tampering with the AMD 80C287 math coprocessor.

AMD 80C287 Math Coprocessor Architecture

From the programmer's view, the AMD 80C287 math coprocessor is a set of additional registers (supplementing those of the 80C286), which include

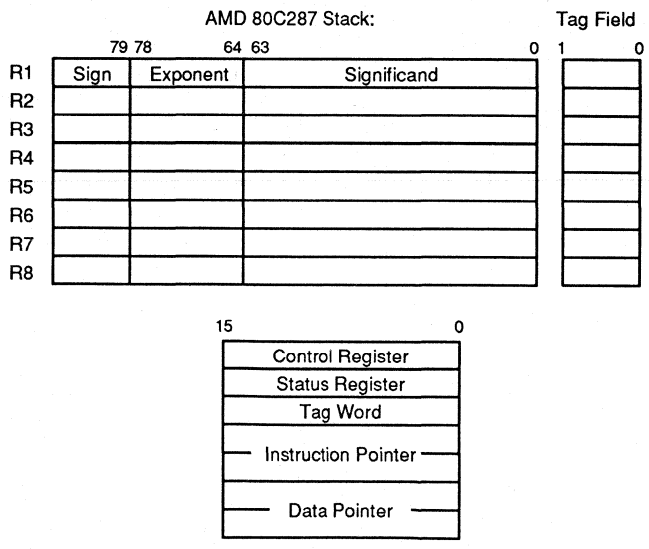
- Eight individually-addressable 80-bit numeric registers, assembled as a register stack.
- Three 16-bit registers consisting of:

  AMD 80C287 status word,
  AMD 80C287 control word,
  Tag word.
- Four 16-bit registers, including the AMD 80C287 instruction and data pointers.

The AMD 80C287 Math Coprocessor Register Stack

Figure 12-2 illustrates the AMD 80C287 register stack. Each of the eight numeric registers in the register stack is 80 bits wide and is split into fields related to the temporary-real data type.

Figure 12-2 **AMD 80C287 Math Coprocessor Register Set**



Numeric instructions address the data registers that involve the register located at the top of the stack. Randomly, this top-of-stack register is specified by the ST (Stack Top) field in the AMD 80C287 status word. ST is decremented by one with load or push operations, which load a value into the new top register. The current ST register's value is stored by a store-and-pop operation, which then increments ST by one. The AMD 80C287 register stack increases down toward lower-addressed registers, similar to the 80C286 stacks in memory.

Many numeric instructions have multiple addressing modes that let the programmer implicitly work on the top of the stack, or explicitly work on those registers involving the ST. Using the expression ST(0), or ST, to represent the current Stack Top and ST(i) to indicate the ith register from ST in the stack ($0 \leq i \leq 7$), 80C286 assemblers maintain these register addressing modes. For instance, if ST includes 011B, where register 3 is the stack top, the subsequent statement would take the sum of the contents of the top two registers on the stack (registers 3 and 5):

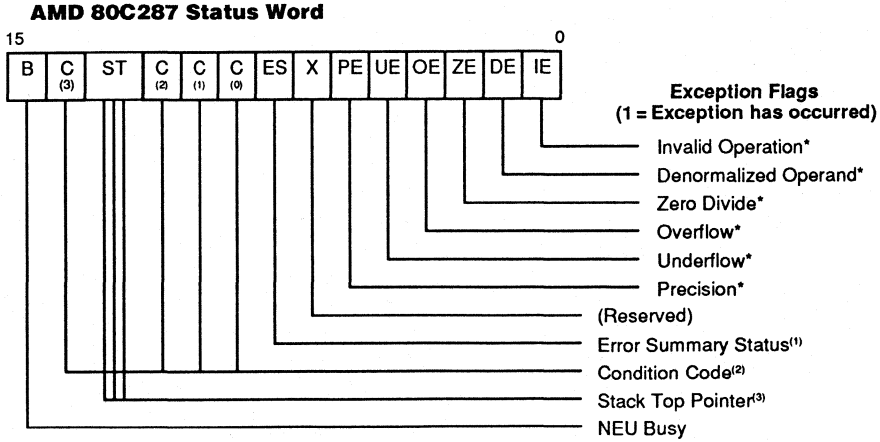
```
FADD   ST,ST(2)
```

By letting routines pass parameters on the register stack, the way the stack is organized and top-relative addressing of the numeric registers ease subroutine programming. Calling routines become more flexible in how they use the stack by using it to pass parameters instead of using dedicated registers. As long as the stack is not completely filled, each routine just loads the parameters onto the stack prior to calling a specific subroutine to execute a numeric calculation. For instance, although ST can refer to physical register 3 in one instance and physical register 5 in another, the subroutine addresses its parameters as ST, ST(1), etc.

The AMD 80C287 Status Word

Figure 12-3 illustrates the 16-bit status word, which expresses the AMD 80C287 device's overall state and may be stored into memory using the FSTSW/FNSTSW,

Figure 12-3 AMD 80C287 Status Word



Notes: 1. ES is set if any unmasked exception bit is set, cleared otherwise.
2. See Table 12-3 for condition code in interpretation.
3. ST Values
000 = Register 0 is top of stack
001 = Register 1 is top of stack
•
•
•
111 = Register 7 is top of stack

* For definitions, see the section on exception handling.

14554A-052

FSTENV/FNSTENV, and FSAVE/FNSAVE instructions. This status word can be moved into the 80C286 AX register with the FSTSW AX/FNSTSW AX instructions, letting the CPU inspect the AMD 80C287 math coprocessor status.

The Busy bit (bit 15) and the $\overline{BUSY}$ pin show if the AMD 80C287 execution unit is idle (B = 0) or if it is performing a numeric instruction or indicating an exception (B = 1). (The Busy bit is not set by instructions FNSTSW, FNSTSW AX, FNSTENV, and FNSAVE, which do not require the Busy bit to be clear for execution.)

The four condition code bits (C3–C2) are like the flags in a CPU in that the AMD 80C287 math coprocessor updates these bits to express the result of arithmetic operations. Table 12-3 outlines the effect of these instructions on the condition code bits, which are mainly used for conditional branching. The FSTSWAX instruction stores the status word into the CPU AX register, letting 80C286 code inspect these condition codes effectively.

Status word bits 12–4 point to the AMD 80C287 register that is the current Stack Top (ST). (See the section on the Register Stack for a discussion on the stack top's importance.)

The six error flags in bits 0–5 of the status word are indicated in Figure 12-3. The error summary status (ES) bit (bit 7) is set if any unmasked exception bits are set;

Table 12-3 Interpreting the AMD 80C287 Condition Codes

InstructionType	C_3	C_2	C_1	C_0	Interpretation
Compare, Test	0	0	X	0	ST > Source or 0 (FTST)
	0	0	X	1	ST < Source or 0 (FTST)
	1	0	X	0	ST = Source or 0 (FTST)
	1	1	X	1	ST is not comparable
Remainder	Q_1	0	Q_0	Q_2	Complete reduction with three low bits of quotient in C_0, C_3, and C_1
	U	1	U	U	Incomplete Reduction
Examine	0	0	0	0	Valid, positive unnormalized
	0	0	0	1	Invalid, positive, exponent = 0
	0	0	1	0	Valid, negative, unnormalized
	0	0	1	1	Invalid, negative, exponent = 0
	0	1	0	0	Valid, positive, normalized
	0	1	0	1	Infinity, positive
	0	1	1	0	Valid, negative, normalized
	0	1	1	1	Infinity, negative
	1	0	0	0	Zero, positive
	1	0	0	1	Empty Register
	1	0	1	0	Zero, negative
	1	0	1	1	Empty Register
	1	1	0	0	Invalid, positive, exponent = 0
	1	1	0	1	Empty Register
	1	1	1	0	Invalid, negative, exponent = 0
	1	1	1	1	Empty Register

Notes: 1. ST = Top of stack.
2. X = Value is not affected by instruction.
3. U = Value is undefined following instruction.
4. Q_n = Quotient bit n following complete reduction ($C_2 = 0$).

otherwise, it is cleared. The $\overline{\text{ERROR}}$ signal is used if this bit is set. Bits 0–5 show if the AMD 80C287 math coprocessor has found one of six possible exception conditions since these status bits were previously cleared or reset.

Control Word

The AMD 80C287 math coprocessor furnishes the programmer with various processing options. To select an option, load a word from memory into the control word. (See Figure 12-4 for the format and encoding of the fields in the control word.)

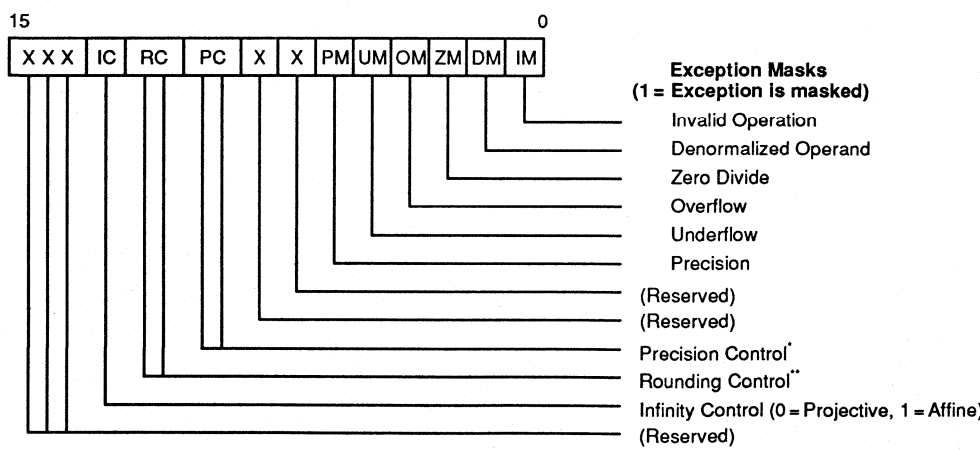
This control word's low-order byte establishes the AMD 80C287 error and exception masking. Bits 0–5 of the control word include separate masks for each of the six exception conditions. The control word's high-order byte establishes the AMD 80C287 processing options, including the following:

- Precision control
- Rounding control
- Infinity control

The Precision control bits (bits 9–8) set the AMD 80C287 internal operating precision below the default precision (64-bit significand). These control bits are compatible with earlier-generation arithmetic processors that have less precision than the AMD 80C287 math coprocessor, as the IEEE 754 standard requires. However, the execution time of numeric calculations is unaffected by setting a lower precision.

The rounding control bits (bits 10–11) supply directed rounding, true chop, and the unbiased round-to-nearest-even mode as the IEEE 754 standard requires.

Figure 12-4 AMD 80C287 Math Coprocessor Control Word Format



*** Precision Control**
00 = 24-Bit Significand
01 = Reserved
10 = 53-Bit Significand
11 = 64-Bit Significand

**** Rounding Control**
00 = Round to Nearest or Even
01 = Round Down (Toward -∞)
10 = Round Up (Toward +∞)
11 = Chop (Truncate Toward Zero)

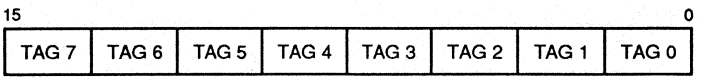
14554A-053

The infinity control bit (bit 12) specifies how the AMD 80C287 math coprocessor regards the special values of infinity. Either affine closure (where positive infinity and negative infinity are different) or projective closure (where infinity is one unsigned quantity) may be indicated. The section on Computation Fundamentals further describes these two alternative views.

The AMD 80C287 Tag Word

The tag word identifies the contents of each register in the register stack, as Figure 12-5 indicates. The AMD 80C287 math coprocessor uses the tag word to track its numeric registers and improve performance. This tag information can be used by programmers to analyze the contents of the numeric registers. The tag values remain in the tag word associated with the physical registers 0–7. In order to associate these tag values with the relative stack registers ST(0) through ST(7), programmers must use the current Stack Top (ST) pointer maintained in the AMD 80C287 status word.

Figure 12-5 **AMD 80C287 Tag Word Format**



Tag Values:
00 = VALID
01 = ZERO
10 = INVALID or INFINITY
11 = EMPTY

The AMD 80C287 Instruction and Data Pointers

Support for programmed exception-handlers is furnished by the AMD 80C287 instruction and data registers. The AMD 80C287 math coprocessor internally saves the instruction address, the operand address (if present), and the instruction opcode each time it performs a math instruction. The FSTENV and FSAVE instructions keep this data in memory, letting exception handlers determine the exact nature of any numeric exceptions that may occur.

Depending on the operating mode of the AMD 80C287 math coprocessor, the instruction and data pointers can be in one of two formats when stored in memory. Pointers stored after an FSTENV instruction are shown in Figure 12-6. In real address mode and formatted like the 8087, these values are the 20-bit physical address and 11-bit opcode. In protected mode, these values are the 32-bit virtual addresses that the program performing the ESC instruction used.

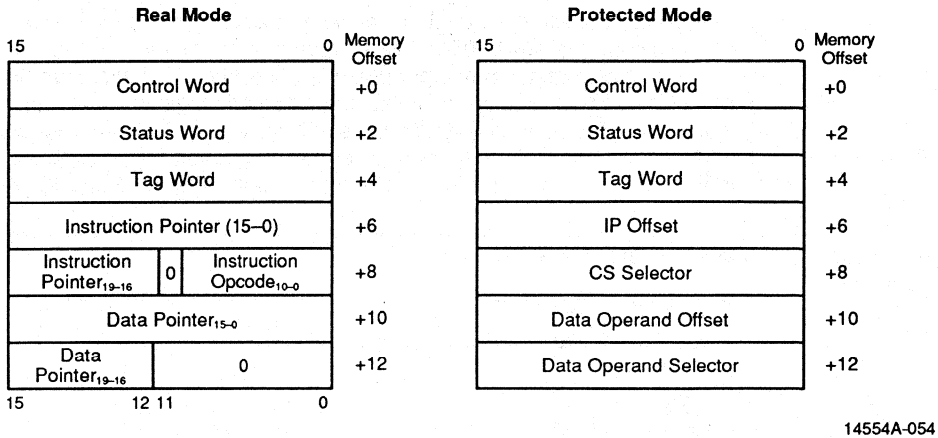
In contrast to the 8087, for which the instruction address pointed only to the ESC instruction opcode, the instruction address maintained in the AMD 80C287 math coprocessor points to any prefixes prior to the instruction.

COMPUTATION FUNDAMENTALS

This section, which describes AMD 80C287 programming concepts known to all applications, discusses the AMD 80C287 math coprocessor's internal number system and the different types of numbers AMD 80C287 programs can use. The most frequently used options for rounding, precision, and infinity (chosen by fields in the control word) are covered, while subsequent sections contain detailed descriptions of

facilities used less frequently. Exception conditions that may occur during AMD 80C287 instruction execution and the options available for handling such exceptions are also discussed.

Figure 12-6 AMD 80C287 Instruction and Data Pointer Image in Memory



14554A-054

Number System

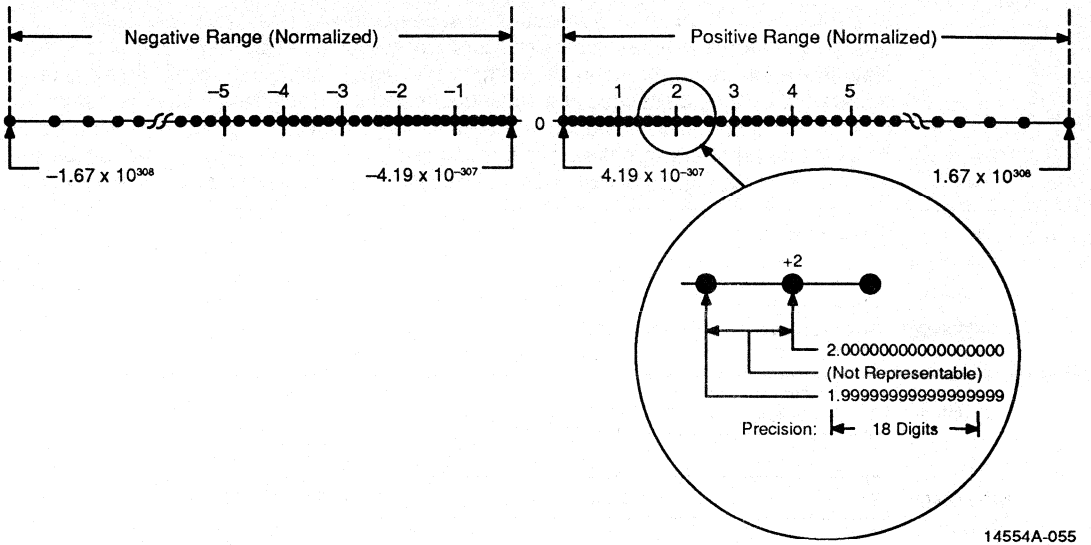
People use a conceptually infinite and continuous system of real numbers for pencil and paper calculations. No upper or lower limit exists for the magnitude of the numbers that can be used in a calculation, or the number of significant digits that the numbers reflect. An infinity of larger and smaller numbers exists whenever any real number is considered. There also exists between any two real numbers an infinity of numbers having more significant digits. For example, 2.51, 2.5897, 2.500001, etc. are between 2.5 and 2.6.

A computer that runs on the entire real number system would be ideal; however, this is not possible. Computers, regardless of their capacity, ultimately have fixed-size registers and memories that restrict the system of numbers that can be used. Both the range and the precision of numbers are set by these restrictions. The result is a fixed and discrete set of numbers, as opposed to infinite and continuous. This sequence, a subset of the real numbers, is meant to produce a helpful approximation of the real number system.

The standard AMD 80C287 real number system is superimposed on a real number line in Figure 12-7. (Figure 12-7 uses decimal numbers for clarity; the AMD 80C287 math coprocessor actually depicts numbers in binary.) The dots specify the subset of real numbers the AMD 80C287 math coprocessor depicts as data and final outcomes of calculations. The AMD 80C287 math coprocessor's range is estimated at $\pm 4.19 \times 10^{-307}$ to $\pm 1.67 \times 10^{308}$. Applications that are needed to deal with data and final outcomes beyond this range are few. The IBM 370's range, for reference, is approximately $\pm 0.54 \times 10^{-78}$ to $\pm 0.72 \times 10^{76}$.

The fixed spacing (see Figure 12-7) indicates how the AMD 80C287 math coprocessor can represent most, but not all, of the real numbers in its range. A gap

Figure 12-7 AMD 80C287 Number System



continually exists between two bordering AMD 80C287 numbers, and the result of a calculation may fall in this space. The AMD 80C287 math coprocessor takes the true result and rounds it to a number it can represent whenever this happens. Therefore, the representation of a real number that needs more digits than the AMD 80C287 math coprocessor can hold (a 20-digit number) is not entirely accurate. The AMD 80C287 math coprocessor's representable numbers are not uniformly distributed on the real number line, either. In fact, there is an equal number of representable numbers between successive powers of 2 (there are as many numbers between 2 and 4 that can be represented as between 65,536 and 131,072). Thus, the gaps between representable numbers are greater as the numbers rise in magnitude. (Note that all integers in the range $\pm 2^{64}$ (about $\pm 10^{18}$) are precisely representable.)

Internally, the AMD 80C287 math coprocessor uses a number system that is a superset of the system in Figure 12-7. The internal format (temporary real) expands the AMD 80C287 math coprocessor's range to approximately $\pm 3.4 \times 10^{-4932}$ to $\pm 1.2 \times 10^{4932}$, and its accuracy to about 19 (equivalent decimal) digits. This format is specified to supply additional range and precision for constants and intermediate results; it is not generally meant for data or final results.

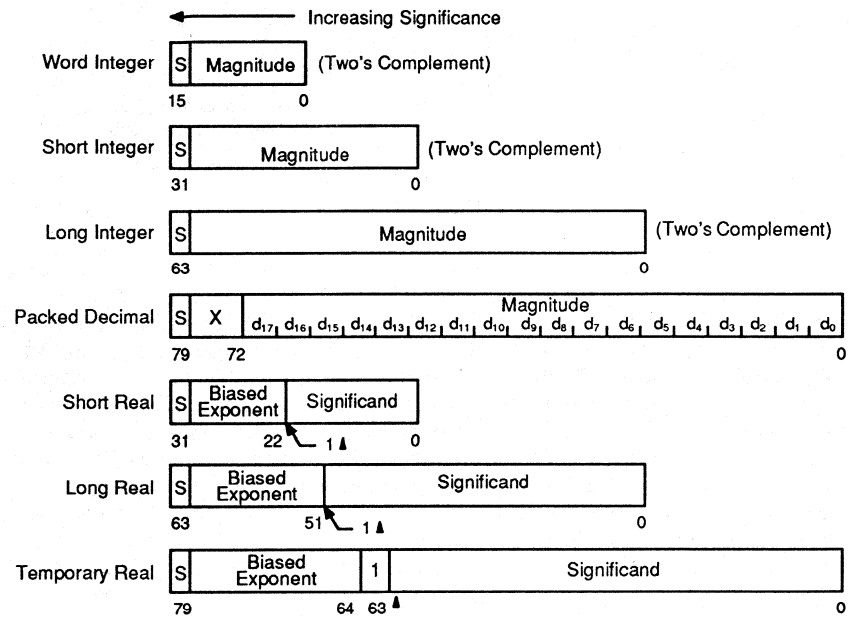
The AMD 80C287 math coprocessor's set of real numbers, from a practical point of view, is adequately large and dense to not restrict the broad majority of microprocessor applications. The AMD 80C287 math coprocessor, when compared to the majority of computers (including mainframes), supplies an excellent approximation of the real number system. Remember, however, that it is not a precise representation, and that arithmetic on real numbers is implicitly approximate.

On the other hand, and just as important, the AMD 80C287 math coprocessor *does* perform precise arithmetic on integer operands (an operation on two integers gives a precise integral result, as long as the true result is an integer and is in range). For instance, $4 \div 2$ returns a precise integer, but $1 \div 3$ does not, and neither does $2^{40} \times 2^{30} + 1$, because the result needs more than 64 bits of precision.

Data Types and Formats

The AMD 80C287 math coprocessor has seven numeric data types, which are organized in three classes: binary integers, packed decimal integers, and binary reals. How these formats are retained in memory (the sign is always found in the highest-addressed byte) is discussed in a subsequent section. Each data type's format is outlined in Figure 12-8. In the figure, the most significant digits of all numbers (and fields inside numbers) are the digits furthest to the left. The range and number of significant (decimal) digits that each format can handle is supplied by Table 12-4.

Figure 12-8 Data Formats



Notes: S = Sign bit (0 = positive, 1 = negative).
dn = Decimal digit (two per byte).
X = Bits have no significance; 80287 ignores when loading, zeros when storing.
▲ = Position of implicit binary point.
1 = Integer bit of significand: stored in temporary real, implicit (always 1) in short and long real.
Exponent Bias (normalized values):
Short Real: 127 (7FH)
Long Real: 1023 (3FFH)
Temporary Real: 16383 (3FFFH)

14554A-056

Table 12-4 Real Number Notation

Notation	Value
Ordinary Decimal	178.125
Scientific Decimal	1Δ78125E2
Scientific Binary	1Δ0110010001E111
Scientific Binary (Biased Exponent)	1Δ0110010001E10000110

	Sign	Biased Exponent	Significand
AMD 80C287 Short Real	0	10000110	01100100010000000000000
			1Δ (implicit)

BINARY INTEGERS

The three binary integer formats are the same, with the exception of length, which controls the range that each format can accommodate. The bit furthest to the left determines the number's sign: 0 = positive and 1 = negative. Negative numbers are depicted in basic two's complement notation (the binary integers are the only AMD 80C287 format to utilize this notation). A positive sign denotes the quantity zero (all bits are 0). The AMD 80C287 word integer format and the 80C286's 16-bit signed integer data type are the same.

DECIMAL INTEGERS

Decimal integers are maintained in packed decimal notation. Two decimal digits are packed into each byte, except the leftmost one, which holds the sign bit (0 = positive, 1 = negative). Negative numbers are not kept in two's complement form; only the sign bit can differentiate them from positive numbers. The leftmost digit is the number's most important digit, and all digits must be in the range 0H–9H.

REAL NUMBERS

The AMD 80C287 math coprocessor maintains real numbers in a three-field binary format, similar to scientific, or exponential, notation. The number's significant digits are kept in the significand field. The exponent field finds the binary point within the significant digits (and thus assesses the number's magnitude), and the sign field shows if the number is positive or negative. (The exponent and significand are equivalent to characteristic and mantissa, two terms that describe floating point numbers on some computers.) Negative numbers can be distinguished from positive numbers only in their significands' sign bits.

As an example, how the real number 178.125 (decimal) is maintained in the short real format of the AMD 80C287 math coprocessor is indicated in Table 12-4, which lists a progression of comparable notations that represent the identical value to show how a number can be transformed from one form to another. Whenever language translators come across programmer-defined real number constants, they perform such a process. Notice that there is not a precise binary equivalent for every decimal fraction. For example, the decimal number $1/10$ cannot be represented precisely in binary (similarly, the number $1/3$ cannot be represented precisely in decimal). A translator generates a rounded binary approximation of the decimal value whenever it comes across such a value.

The AMD 80C287 math coprocessor generally keeps the digits of the significand in normalized form. In other words, with the exception of the value zero, the significand is an integer and a fraction as indicated below:

$1\Delta fff...ff$

where Δ specifies an assumed binary point. Depending on the real format, the number of fraction bits is varied: 23 for short, 52 for long, and 63 for temporary real. Leading zeros in small values ($|X| < 1$)are eliminated by the AMD 80C287 coprocessor normalizing real numbers so their integer bit is always 1. This process optimizes the number of significant digits that a significand of a specific width can accommodate. Note: In the short and long real formats, the integer bit, being implicit, is not actually stored; it is physically maintained in the temporary real format only.

If just the significand with its assumed binary point was scrutinized, the value of all normalized real numbers would be between 1 and 2. The exponent field finds the actual binary point in the significant digits. Like decimal scientific notation, a positive exponent shifts the binary point to the right, and a negative exponent shifts the binary point to the left, placing leading zeros where needed. The position of the assumed

binary point and the position of the actual binary point are the same, when the unbiased exponent is zero. So the exponent field establishes the magnitude of a real number.

The AMD 80C287 math coprocessor maintains exponents in a biased form to make it easier to compare real numbers (for sorting). In other words, a constant is added to the true exponent described earlier. This bias' value differs for each real format (see Figure 12-8), and has been selected in order to force the biased exponent to be a positive value. Thus, two real numbers (of identical format and sign) can be compared as if they are unsigned binary integers. When comparing them bitwise from left to right, starting with the exponent bit furthest to the left, the first different bit position orders the numbers and the comparison need not continue. To determine a number's true exponent, just subtract the bias value of its format.

There are short and long real formats in memory only. If a number in one of these formats is loaded into an AMD 80C287 register, it automatically becomes temporary real format, which is utilized for all internal operations. Similarly, data in registers can be changed to short or long real for storage in memory. Intermediate results that cannot be held in registers can be stored in temporary real format in memory,as well.

To maintain real number data and results, the majority of applications should use the long real form, since it supplies adequate range and precision to generate accurate results with little programmer attention. The short real format is suitable for applications restricted by memory, but this format has a smaller safety margin. The short real format also aids in debugging algorithms, because roundoff problems will appear quicker in this format. The temporary real format, which should usually be used for storing intermediate results, loop accumulations, and constants, has extra length that is designed to protect final results from the effects of rounding and overflow/underflow in intermediate calculations. The range and precision of the long real form, however, are sufficient for the majority of microcomputer applications.

Rounding Control

The AMD 80C287 math coprocessor internally utilizes three additional bits (guard, round, and sticky bits) to represent the infinitely exact true result of a computation (programmers cannot access these bits). The AMD 80C287 math coprocessor delivers the infinitely exact true result whenever the destination can represent it. In arithmetic and store operations, rounding results whenever the destination's format cannot precisely represent the infinitely exact true result. A real number, for instance, may be rounded if it is maintained in a shorter real format or in an integer format, or the infinitely exact true result may be rounded when it returns to a register.

The AMD 80C287 math coprocessor has four rounding modes, which can be selected by the RC field in the control word (see Figure 12-4). Given a true result *b* that the target data type is unable to represent, the AMD 80C287 math coprocessor decides on the two representable numbers *a* and *c* that most closely surround *b* in value ($a < b < c$). Based on the mode selected by the RC field, as Table 12-5 indicates, the processor then rounds (changes) *b* to *a* or to *c*. In a result, round then presents an error that is less than one unit in the last place to which the result is rounded. Round to Nearest, the default mode, is appropriate for the majority of applications because it furnishes the most correct and statistically unbiased approximation of the true result. The chop mode is furnished for integer arithmetic applications. Round Up and Round Down, termed directed rounding, can impose interval arithmetic. Interval arithmetic produces a justifiable result that is distinct from rounding and other errors. To

calculate the upper and lower bounds of an interval, perform an algorithm twice, rounding up in one pass and down in the other.

Table 12-5 Rounding Modes

RC Field Rounding Action		Rounding Mode
00	Round to nearest	Closer to *b* of *a* or *c*; if equally close, select even number (the one whose least significant bit is zero).
01	Round down (toward $-\infty$)	*a*
10	Round up (toward $+\infty$)	*c*
11	Chop (toward 0)	Smaller in magnitude of *a* or *c*

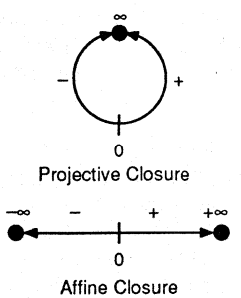
Precision Control

The AMD 80C287 math coprocessor calculates results with either 64, 53, or 24 bits of precision in the significand, as determined by the precision control (PC) field of the control word. The default setting, appropriate for most applications, is the full 64 bits of significance that the temporary-real format provides. The proposed IEEE standard requires the other settings, which are supplied in order to become compatible with the specifications of particular programming languages that already exist. Specifying less precision cancels the benefits of the temporary real format's expanded fraction length, and does not boost performance speed. The rounding of the fractional value makes the unused bits on the right zeros whenever less precision is specified.

Infinity Control

One of two models of infinity may close the AMD 80C287 math coprocessor's system of real numbers: projective and affine closure. Figure 12-9 schematically shows these two ways to close the number system. The IC field's setting in the control word chooses from the two models. Projective is the default means of closure, and is suggested for most computations. Under projective closure, the AMD 80C287 math coprocessor regards the special values $+\infty$ and $-\infty$ as one unsigned infinity (just like its treatment of signed zeros). The AMD 80C287 math coprocessor recognizes the signs of $+\infty$ and $-\infty$ in the affine mode.

Figure 12-9 Projective versus Affine Closure



14554A-057

Although affine mode supplies more information than projective, instances when the sign may actually denote misinformation do occur. Take, for example, an algorithm that returns an intermediate result x of +0 and -0 (the identical numeric value) in separate executions. If $1/x$ were then calculated in affine mode, two completely different values ($+\infty$ and $-\infty$) would result from the same numeric values of x. Conversely, projective mode furnishes less information, yet never generates misinformation. Basically, thus, projective mode should be most commonly used, with affine mode available for local computations in which the programmer can benefit from the sign and knows absolutely that the nature of the computations will not return a deceptive result.

SPECIAL COMPUTATIONAL SITUATIONS

Apart from its ability to represent positive and negative numbers, the AMD 80C287 data formats can describe other entities. These special values supply additional flexibility, but the majority of users do not need to understand them to successfully utilize the AMD 80C287 math coprocessor. The special values that can result in particular cases and the significance of each are discussed in this section, as are the 80C286 exceptions for writers of exception handlers and anyone interested in exploring AMD 80C287 math coprocessor computation limits.

Programmers who write exception handlers will find the material presented in this section of most interest. Other readers may want to just browse this section.

Special Numeric Values

The AMD 80C287 math coprocessor data formats include encodings for multiple special values, as well as the common real or integer data values that normal calculations yield. These special values are important, and they can convey pertinent information about the computations or operations that generated them. The types of special values are listed below:

- Non-normal real numbers, such as

  denormals
  unnormals
- Zeros and pseudo zeros
- Positive and negative infinity
- NaN (Not-a-Number)
- Indefinite

The origins and significance of each of these special value types are described below. The end of this section (see Tables 12-11 through 12-14) supplies a complete description of the encoding process of each of these special values for every numeric data type.

NONNORMAL REAL NUMBERS

The AMD 80C287 math coprocessor, as discussed earlier, usually maintains nonzero real numbers in normalized floating-point form; in other words, the integer (leading) bit of the significand remains a 1. This bit is explicitly kept in the temporary real format, and is implicitly interpreted to be a one (1Δ) in the short- and long-real formats. Normalized storage lets the greatest number of significant digits be held in a significand of a certain width because leading zeros are eliminated.

As a floating-point numeric value gets closer to zero, normalized storage cannot represent the value precisely. The AMD 80C287 math coprocessor, on such occasions, is able to store and operate on reals that are not normalized (i.e., reals whose significands include at least one leading zero). Nonnormals usually occur when the outcome of a calculation returns a value that is too small to be expressed in normal form.

Nonnormal values can be found in one of the two following forms:

- The floating-point exponent may be maintained at its most negative value (a Denormal)
- The integer bit (and possibly other leading bits) of the significand may be zero (an Unnormal)

Aside from losing some precision (the leading zeros decrease the number of significant bits), the leading zeros of nonnormals let smaller numbers be expressed. In common algorithms, very small values are most likely to return as intermediate, as opposed to final, results. Values as small as $\pm 3.4 \times 10^{-4932}$ can be expressed by utilizing the AMD 80C287 temporary real format for holding intermediate; thus, the occurrence of nonnormal numbers is an unusual phenomenon in AMD 80C287 applications. Notwithstanding, the AMD 80C287 math coprocessor is able to load, maintain, and operate on nonnormalized real numbers if they should occur.

Denormals and Gradual Underflow

A denormal results from the AMD 80C287 math coprocessor's response to an underflow exception when the programmer has masked that exception (see Figure 12-4, the AMD 80C287 control word). Underflow results when the absolute value of a real number is so small it cannot be expressed in the destination format (that is, when the true result's exponent is too negative for the destination format to represent). Underflow will occur if, for instance, there is a true exponent of –130 and the destination is short real, because –126 is the smallest exponent this format can handle. If the destination were long real or temporary real, however, underflow would not result, since these formats can accommodate exponents down to –1023 and –16,383, respectively.

The majority of computers underflow abruptly. That is, they just yield a zero result, which will probably generate an objectionable final result if computation persists. When the underflow exception is masked, however, the AMD 80C287 math coprocessor underflows gradually. By denormalizing the result until it is just inside the destination format's exponent range, gradual underflow is achieved. Denormalizing is the process of increasing the true result's exponent and placing a comparable leading zero in the significand, moving the remainder of the significand one place to the right. Denormal values are possible in any short-real, long-real, or temporary-real formats. The manner in which a result could be denormalized to fit a short-real destination is shown in Table 12-6.

The AMD 80C287 masked response to underflow lets computation proceed and avoids program intervention, at the same time causing an error that has about an equal risk of infecting the final result as roundoff error. Roundoff (precision) errors often occur in real number calculations. They may ruin the result of computation, but frequently they do not. Accepting that roundoff errors are typically nonfatal, computation routinely continues, and the programmer checks the final results to determine if these errors have had an important effect. The masked underflow response of the AMD 80C287 math coprocessor lets programmers handle underflows in a comparable manner. As the computation proceeds, the programmer can probe

Table 12-6 **Denormalization Process**

Operation	Sign	Exponent*	Significand
True Result	0	−129	1Δ01011100...00
Denormalize	0	−128	0Δ101011100...00
Denormalize	0	−127	0Δ0101011100...00
Denormalize	0	−126	0Δ00101011100...00
Denormalize Result**	0	−126	0Δ00101011100...00

* Expressed as unbiased, decimal number.
** Before storing, significand is rounded to 24 bits, integer bit is dropped, and exponent is biased by adding 126.

the final result to decide if an underflow has had consequences of any significance. (If such an effect has occurred, an invalid operation will most likely be signaled thereafter in the computation.)

A denormal or a zero is created in the process of denormalization. The exponents of denormals are always the smallest for their formats; thus, denormals are easy to identify. In biased form, this is always the bit string: 00...00. The zeros also have this same exponent value; a denormal, however, has a nonzero significand. Denormals in registers are labeled special. Tables 12-13 and 12-14 illustrate how denormal values are encoded for every real data format.

The low-order significand bits may be lost as they are moved off the right in the process of denormalization. In an extreme instance, the leading zeros replace all the true result's significand bits and the denormalization results in a true zero. If the value is within a register, it is so labeled.

Note: This is a relatively unusual occurrence and, at any rate, is no more troublesome than abrupt underflow.

Denormals rarely occur in most applications. Common debugged algorithms produce particularly small results while intermediate subexpressions are evaluated; the end result is typically of a suitable size for its short or long real destination. If intermediate results are stored in temporary real, as encouraged, underflow is very implausible because of this format's vast range. Only when an application produces so many intermediates that they cannot be stored either on the register stack or in temporary real memory variables will a denormal probably occur. Underflow may occur if storage restrictions require short or long reals to be used for intermediates, and small values are generated; denormals may occur if underflow is masked.

Table 12-7 illustrates how accessing a denormal may result in an exception (the denormalized exception indicates that a denormal has been retrieved). The significance of denormals may be less because of lost low-order bits, and an option of the proposed IEEE standard disallows operations on nonnormalized operands. An exception handler responding to unmasked denormalized exceptions may be an implementation of this option. The majority of users mask this exception so computation can continue; the user will determine if any precision was lost when the final result is delivered.

The division and remainder operations do not recognize denormal divisors, invoking the invalid operation exception (see Table 12-7). Also remember that normalized operands are necessary for transcendental instructions, which do *not* inspect for exceptions. Under any other circumstances, the AMD 80C287 math coprocessor transforms denormals to unnormals, and the rules concerning unnormal arithmetic then go into effect. (See the subsequent section for a description of unnormals.)

Table 12-7 **Exceptions Due to Denormal Operands**

Operation	Exception	Masked Response
FLD (short/long real)	D	Load as equivalent unnormal
Arithmetic (except following)	D	Convert (in a work area) denormal to equivalent unnormal and proceed
Compare and test	D	Convert (in a work area) denormal to equivalent unnormal and proceed
Division or FPREM with denormal divisor	I	Return real indefinite

Unnormals—Descendents Of Denormal Operands

As the result of a computation using denormal operands, an unnormal is, thus, the descendent of the AMD 80C287 math coprocessor's masked underflow response. Only in the temporary real format can an unnormal exist. Although an unnormal may have any exponent of a normal value (any nonzero value in biased form), the integer bit of its significand (always 0) differentiates it from a normal. Within a register an unnormal is labeled valid. Unnormals differ from denormals; the latter have an exponent of 00...00 in biased form.

Unnormals let arithmetic proceed after an underflow, yet they still retain their identity as numbers that may have less significance. In other words, unnormal operands produce unnormal results, assuming that their unnormality significantly affects the result. Thus, unnormals are prevented from posing as normals, or numbers with complete significance. Conversely, if an unnormal insignificantly affects a calculation with a normal, the outcome is normal. For example, taking the sum of a small unnormal and a large normal returns a normal result; the opposite situation, an unnormal.

The way the instruction set handles unnormal operands is illustrated in Table 12-8.

Note: The unnormal is possibly the initial operand or a temporary one produced by the AMD 80C287 math coprocessor from a denormal.

ZEROS AND PSEUDO ZEROS

In the real and decimal integer formats, the value zero can be positive or negative, but a binary integer zero stays positive. For the purpose of computation, the value of zero always acts the same, despite its sign. Generally, the programmer is aware that a zero may be signed. A zero's sign can be determined by the FXAM instruction, if needed.

The zeros described above are true zeros (i.e., if one of them is loaded or produced in a register, the register is labeled zero). The results of instructions performed with zero operands are listed in Table 12-9, which indicates how nonzero operands can generate a true zero, as well.

A special class of values, pseudo zeros, is included in the temporary real format only. A pseudo zero is an unnormal that has a significand of all zeros; its (biased) exponent is nonzero, whereas true zeros have a zero exponent. A pseudo zero's exponent cannot be all ones, either, because this encoding is kept for infinities and NANs. A pseudo zero outcome will be generated by taking the product of two unnormals with more than 64 total leading zero bits in their significands. Although this is a rare occurrence in the majority of applications, it is possible.

Table 12-8 **Unnormal Operands and Results**

Operation	Result
Addition/subtraction	Normalization of operand with larger absolute value determines normalization of result.
Multiplication	If either operand is unnormal, result is unnormal.
Division (unnormal dividend only)	Result is unnormal.
FPREM (unnormal dividend only)	Result if normalized.
Division/FPREM (unnormal divisor)	Signal invalid operation.
Compare/FTST	Normalize as much as possible before making comparison.
FRNDINT	Normalize as much as possible before rounding.
FSQRT	Signal invalid operation.
FST, FSTP (short/long real destination)	If value is above destination's underflow boundary, then signal invalid operation; else signal underflow.
FSTP (temporary real destination)	Store as usual.
FIST, FISTP, FBSTP	Signal invalid operation.
FLD	Load as usual.
FXCH	Exchange as usual.
Transcendental instructions	Undefined; operands must be normal and are not checked.

Pseudo zero operands act like unnormals, with the exception of the following cases where they generate the identical results of true zeros:

- Compare and verify instructions
- FRNDINT (round to integer)
- Divide, with the dividend either a true zero or a pseudo zero (the divisor is a pseudo zero)

Except for establishing the result's sign, the pseudo zero(s) acts like unnormals when adding and subtracting a pseudo zero and a true zero or another pseudo zero. Table 12-9 indicates how the sign for two true zero operands is established.

INFINITY

The real formats accommodate signed representations of infinities, and these values are encoded with a biased exponent of all ones and a significand of 1Δ00...00. The infinity is labeled special if it is within a register. The significand can differentiate infinities from NANs, even real indefinite.

An infinity may be coded by a programmer or created by the AMD 80C287 math coprocessor as its masked response to an overflow or a zero divide exception. Note: When rounding is up or down, the masked response can produce the greatest valid value that can be expressed in the destination instead of infinity (see Table 12-10).

Table 12-9 Zero Operands and Results

Operation/Operands	Result
FLD,FBLD[1]	
+0	+0
−0	−0
FILD[2]	
+0	+0
FST, FSTP	
+0	+0
−0	−0
+X[3]	+0
−X[3]	−0
FBSTP	
+0	+0
−0	−0
FIST, FISTP	
+0	+0
−0	+0
+X[4]	+0
−X[4]	+0
Addition	
+0 plus +0	+0
−0 plus −0	−0
+0 plus −0, −0 plus +0	*0[5]
−X plus +X, +X plus −X	*0[5]
+0 plus +X, +X plus +0	**X[6]
Subtraction	
+0 minus −0	+0
−0 minus +0	−0
+0 minus +0, −0 minus −0	*0[5]
+X minus +X, −X minus −X	*0[5]
±0 minus ±X, ±X minus ±0	**X[6]
	F2XM1
Multiplication	
+0 · +0, −0 · −0	+0
+0 · −0, −0 · +0	−0
+0 · +X, +X · +0	+0
+0 · −X, −X · +0	−0
−0 · +X, +X · −0	−0
−0 · −X, −X · −0	+0
+X · +Y, −X · −Y	+0, underflow[7]
+X · −Y, −X · +Y	−0, underflow[7]

Operation/Operands	Result
Division	
±0 ÷ ±0	Invalid operation
±X ÷ ±0	Zero divide
+0 ÷ +X, −0 ÷ −X	+0
+0 ÷ −X, −0 ÷ +X	−0
−X ÷ −Y, +X ÷ +Y	+0, underflow[8]
−X ÷ +Y, +X ÷ −Y	−0, underflow[8]
FPREM	
±0 rem ±0	Invalid operation
±X rem ±0	Invalid operation
+0 rem +X, +0 rem −X	+0
−0 rem +X, −0 rem −X	−0
+X rem +Y, +X rem −Y	+0[9]
−X rem −Y, −X rem +Y	−0[9]
FSQRT	
−0	−0
+0	+0
Compare	
+0 : +X	A < B
+0 : +0	A = B
+0 : −X	A > B
FTST	
±0	Zero
FCHS	
+0	−0
−0	+0
FABS	
±0	+0
+0	+0
−0	−0
FRNDINT	
+0	+0
−0	−0
FXTRACT	
+0	Both +0
−0	Both −0

Notes: 1. Arithmetic and compare operations with real memory operands interpret the memory operand signs in the same way.
2. Arithmetic and compare operations with binary integers interpret the integer sign in the same manner.
3. Severe underflows in storing to short or to long real may generate zeros.
4. Small values (|X| < 1) stored into integers may round to zero.
5. Sign is determined by round mode:
   * = + for nearest, up, or chop
   * = − for down.
6. ** = sign of X.
7. Very small values of X and Y may yield zeros, after rounding of true result. AMD 80C287 math coprocessor signals underflow to warn that zero has been yielded by nonzero operands.
8. Very small X and very large Y may yield zero, after rounding of true result. AMD 80C287 math coprocessor signals underflow to warn that zero has been yielded from nonzero operands.
9. When Y divides into X exactly.

Table 12-10 **Masked Overflow Response with Directed Rounding**

True Result Normalization	Sign	Rounding Mode	Result Delivered
Normal	+	Up	+∞
Normal	+	Down	Largest finite positive number*
Normal	–	Up	Largest finite negative number*
Normal	–	Down	–∞
Unnormal	+	Up	+∞
Unnormal	–	Down	Largest exponent, result's significand**
Unnormal	+	Up	Largest exponent, result's significand**
Unnormal	–	Down	–∞

* The largest valid representable reals are encoded:
exponent: 11...10B
significand: (1)Δ11...10B

** The significand retains its identity as an unnormal; the true result is rounded as usual (effectively chopped toward 0 in this case). The exponent is encoded 11...10B.

As operands, the behavior of infinities varies, depending on how the infinity control field in the control word is set (see Table 12-11). When the projective model of infinity is chosen, the infinities act as one unsigned expression; thus, infinity can be compared only with itself. The signs of the infinities are recognized in affine mode, making it possible to compare.

NOT A NUMBER (NaN)

A NaN, part of a group of special values that exist only in the real formats, has an exponent of 11..11B. It can be positive or negative, and have any significand except 1Δ00..00B, which is given to the infinities. A NaN within a register is labeled special.

The AMD 80C287 math coprocessor produces the special NaN, real indefinite, as its masked response to an invalid operation exception. This NaN is negative, and its significand is 1Δ100..00. Other NaNs express values produced by programmers.

The AMD 80C287 math coprocessor indicates an invalid operation exception in its status word whenever it utilizes a NaN operand. The AMD 80C287 math coprocessor's masked exception response yields the NaN as the operation result if this exception is masked in the AMD 80C287 math coprocessor control word. If both of an instruction's operands are NaNs, the NaN with the greater absolute value results. Thus, a NaN entering a computation increases throughout the computation and ultimately becomes the end result.

Note: The transcendental instructions do not inspect their operands, and thus a NaN generates a result that is not defined.

The programmer, by unmasking the invalid operation exception, can utilize NaNs to trap to the exception handler. This approach's general nature and the great number of available NaN values supply the advanced programmer with a tool applicable to various special situations.

A compiler, for example, may utilize NaNs in reference to uninitialized (real) array elements. It is possible for the compiler to preinitialize each array element with a NaN whose significand included the element's index, or relative position. An application program trying to access an uninitialized element would utilize the NaN the compiler provided. An unmasked invalid operation exception would result in an interrupt and activate the exception handler, which could then establish which element had been

Table 12-11 Infinity Operands and Results

Operation	Projective Result	Affine Result
Addition		
+∞ plus +∞	Invalid operation	+∞
−∞ plus −∞	Invalid operation	−∞
+∞ plus −∞	Invalid operation	Invalid operation
−∞ plus +∞	Invalid operation	Invalid operation
±∞ plus ±X	*∞	*∞
±X plus ±∞	*∞	*∞
Subtraction		
+∞ minus −∞	Invalid operation	+∞
−∞ minus +∞	Invalid operation	−∞
+∞ minus +∞	Invalid operation	Invalid operation
−∞ minus −∞	Invalid operation	Invalid operation
±∞ minus ±X	*∞	*∞
±X minus ±∞	**∞	**∞
Multiplication		
±∞ · ±∞	⊕	⊕
±∞ · ±Y	⊕	⊕
±X · ±∞, ±∞ · ±0	Invalid operation	Invalid operation
Division		
±∞ ÷ ±∞	Invalid operation	Invalid operation
±∞ ÷ ±X	⊕	⊕
±X ÷ ±∞	⊕	⊕
FSQRT		
−∞	Invalid operation	Invalid operation
+∞	Invalid operation	+∞
FPREM		
±∞ rem ±∞	Invalid operation	Invalid operation
±∞ rem ±X	Invalid operation	Invalid operation
±Y rem ±∞	*Y	*Y
±0 rem ±∞	*0	*0
FRNDINT		
±∞	*∞	*∞
FSCALE		
±∞ scaled by ±∞	Invalid operation	Invalid operation
±∞ scaled by ±X	*∞	*∞
±0 scaled by ±∞	*0	*0
±Y scaled by ±∞	Invalid operation	Invalid operation
FXTRACT		
±∞	Invalid operation	Invalid operation
Compare		
±∞: ±∞	A = B	−∞ < +∞
±8: ±Y	A?B (and) invalid operation	−∞ < Y < +∞
±8: ±0	A?B (and) invalid operation	−∞ < 0 < +∞
FTST		
±∞	A?B (and) invalid operation	*∞

X = zero or nonzero operand.
Y = nonzero operand.
* = sign of original operand.
** = sign is complement of original operand's sign.
⊕ = sign is Exclusive OR original operand signs (+ if operands had same sign, _ if operands had different signs) original.

accessed. This is possible because the exception pointers' operand address field would point to the NaN, which would include the array element's index number.

Speeding up the debugging process is possible with NaNs. Several errors are usually present in a program's early testing phase. An exception handler could be written to store diagnostic information in memory whenever it was activated. After the storage process, the exception handler could return a NaN as the result of the erroneous instruction. The NaN could then point to its corresponding diagnostic area in memory. The program would proceed, producing an individual NaN for each error. At the program's conclusion, the NaN results could access the diagnostic data stored when the errors occurred, and thus, several errors could be assessed and fixed in a single test run.

INDEFINITE

One distinct encoding is reserved to express the special value indefinite for each AMD 80C287 math coprocessor numeric data type. This encoding is the AMD 80C287 math coprocessor's reaction to a masked invalid-operation exception. When involving reals, the indefinite value can be saved and loaded like any NaN, but always keeps its special identity (programmers are urged to use this encoding only for this purpose). The AMD 80C287 math coprocessor can save packed decimal indefinite in an FBSTP instruction. An undefined result will occur if an attempt is made to use this encoding in an FBLD instruction. In the binary integers, such encoding may express either indefinite or the greatest negative number the format can support (-2^{15}, -2^{31}, or -2^{63}). This encoding is stored as the AMD 80C287 math coprocessor's masked response to an invalid operation, or else it is stored whenever a source register's value represents or rounds to the greatest negative integer the destination can express. Whenever its origin is unclear, check the invalid operation exception flag and see if an exception response generated the value. This encoding is continually regarded as a negative number, whenever it is loaded, or used by an integer arithmetic or compare operation. Thus, a packed decimal or binary integer cannot load indefinite.

ENCODING OF DATA TYPES

Tables 12-12 through 12-15 illustrate how each of the special values discussed above is encoded for every numeric data type. The least-significant bits, indicated at the right in these tables, are saved in the lowest memory addresses, and the bit furthest to the left of the highest-addressed byte is always the sign bit.

Numeric Exceptions

The AMD 80C287 math coprocessor acknowledges a numeric exception condition whenever the AMD 80C287 math coprocessor tries to execute a numeric operation with invalid operands or generates an unrepresentable result. The AMD 80C287 math coprocessor looks for the six classes of exceptions listed below during execution of numeric instructions:

1. Invalid operation
2. Divide-by-zero
3. Denormalized operand
4. Numeric overflow
5. Numeric underflow
6. Inexact result (precision)

Table 12-12 Binary Integer Encodings

Class		Sign	Magnitude
Positives	(Largest)	0	11...11
		•	•
		•	•
		•	•
	(Smallest)	0	00...01
	Zero	0	00...00
Negatives	(Smallest)	1	11...11
		•	•
		•	•
		•	•
	(Largest/indefinite*)	1	00...00
		Word:	← 15 bits →
		Short:	← 31 bits →
		Long:	← 63 bits →

Note: If this encoding is used as a source operand (as in an integer load or integer arithmetic instruction), the AMD 80C287 math coprocessor interprets it as the largest negative number representable in the format: -2^{15}, -2^{31}, or -2^{63}. The AMD 80C287 math coprocessor will deliver this encoding to an integer destination in two cases:
1. If the result is the largest negative number.
2. As the response to a masked invalid operation exception, in which case it represents the special value integer indefinite.

Table 12-13 Packed Decimal Encodings

Class		Sign		Magnitude			
				digit	digit	digit	digit . . . digit
Positives	(Largest)	0	0000000	1001	1001	1001	1001 . . . 1001
		•	•				•
		•	•				•
		•	•				•
	(Smallest)	0	0000000	0000	0000	0000	0000 . . . 0000
	Zero	0	0000000	0000	0000	0000	0000 . . . 0000
	Zero	1	0000000	0000	0000	0000	0000 . . . 0000
Negatives	(Smallest)	1	0000000	0000	0000	0000	0000 . . . 0001
		•	•				•
		•	•				•
		•	•				•
	(Largest)	1	0000000	1001	1001	1001	1001 . . . 1001
	Indefinite[1]	1	1111111	1111	1111	UUUU[2]	UUUU . . . UUUU
		← 1 byte →		← 9 bytes →			

Notes 1. The packed decimal indefinite encoding is stored by FBSTP in response to a masked invalid operation exception. Attempting to load this value via FBLD produces an undefined result.
2. UUUU means bit values are undefined and may contain any value.

Table 12-14 Real and Long Real Encodings

Class			Sign	Biased Exponent	Significand* Δff...ff
Positives		NaNs	0 • • • 0	11...11 • • • 11...11	11...11 • • • 00...01
		∞	0	11...11	00...00
	Reals	Normals	0 • • • 0	11...10 • • • 00...01	11...11 • • • 00...00
		Denormals	0 • • • 0	00...00 • • • 00...00	11...11 • • • 00...01
		Zero	0	00...00	00...00
Negatives		Zero	1	00...00	00...00
		Denormals	1 • • • 1	00...00 • • • 00...00	00...01 • • • 11...11
		Normals	1 • • • 1	00...01 • • • 11...10	00...00 • • • 11...11
		∞	1	11...11	00...00
	NaNs		1 • • •	11...11 • • •	00...01 • • •
		Indefinite	1	11...11	10...00
			• • • 1	• • • 11...11	• • • 11...11
			Short: Long:	← 8 bits → ← 11 bits →	← 23 bits → ← 52 bits →

*Integer bit is implied and not stored.

Table 12-15 Temporary Real Encodings

Class			Sign	Biased Exponent	Significand* 1.ff...ff
Positives	NaNs		0 • • • 0	11...11 • • • 11...11	111...11 • • • 100...01
	∞		0	11...11	100...00
	Reals		0 • • • 0	11...10 • • • •	Normals 111...11 • • • 100...00
			0 0 • • • 0	• • • • • 00...01	Unnormals 011...11 • • • 000...01
Negatives			0 • • • 0	00...00 • • • 00...00	Denormals 011...11 • • • 000...01
		Zero	0	00...00	000...00
		Zero	1	00...00	000...00
			1 • • • 1	00...00 • • • 00...00	Denormals 000...01 • • • 011...11
			1 • • • • • •	00...01 • • • • • •	Unnormals 000...00 • • • 011...11
			• • • • • • 1	• • • • • • 11...10	Normals 100...00 • • • 111...11

Table 12-15 Temporary Real Encodings, continued

Class			Sign	Biased Exponent	Significand* 1.ff...ff
	∞		1	11...11	100...00
Negatives	NaNs		1 • • •	11...11 • • •	100...00 • • •
		Indefinite	1	11...11	110...00
			• • • 1	• • • 11...11	• • • 111...11
				← 15 bits →	← 64 bits →

INVALID OPERATION

Should any of the following occur, the AMD 80C287 math coprocessor indicates an invalid operation:

- An effort to load a register that is not empty (stack overflow).
- An effort to pop an operand from an empty register (stack underflow).
- An operand is a NaN.
- The operands make the operation indeterminate (square root of a negative number, 0/0).

A program error typically results from an invalid operation.

ZERO DIVISOR

The AMD 80C287 math coprocessor indicates a zero divide exception if an instruction tries to divide a finite nonzero operand by zero.

DENORMALIZED OPERAND

The AMD 80C287 math coprocessor indicates the denormalized operand exception if an instruction tries to operate on a denormal. Such an exception lets users produce in software an option of the proposed IEEE standard that states that operands must be prenormalized prior to their use.

NUMERIC OVERFLOW AND UNDERFLOW

The AMD 80C287 math coprocessor indicates a numeric overflow if a numeric result's exponent is too great for the destination real format. On the other hand, if a result's exponent is too small to be expressed in the destination format, a numeric underflow is indicated. The operation's result, if either exception occurs, is beyond the range of the destination real format.

Normal algorithms usually generate very large and small numbers in the calculation of intermediate, versus the end, results. Overflow and underflow are relatively unusual occurrences in the majority of AMD 80C287 math coprocessor applications due to the large range of the temporary real format (suggested as intermediates' destination format).

INEXACT RESULT

If an operation's outcome cannot be precisely represented in the destination format, the AMD 80C287 math coprocessor rounds the number and indicates the precision exception. The fraction 1/3, for instance, is not exactly representable in binary form. This frequent exception points to the loss (typically allowable) of some accuracy, and is supplied for applications that must execute precise arithmetic only.

HANDLING NUMERIC ERRORS

The AMD 80C287 math coprocessor selects one of two possible paths to handle numeric errors:

- The AMD 80C287 math coprocessor can manage the error itself, thereby generating the most reasonable result and letting numeric program execution proceed without interruption.
- The CPU can activate a software exception handler to manage the error.

All six exception conditions discussed earlier have a comparable flag bit in the AMD 80C287 math coprocessor status word and a mask bit in the AMD 80C287 math coprocessor control word. The AMD 80C287 math coprocessor takes suitable default action for a masked exception (the comparable mask bit in the control word = 1) and the computation proceeds. In the case of an unmasked exception (mask = 0), the AMD 80C287 math coprocessor sends the $\overline{\text{ERROR}}$ output to the 80C286 to indicate the exception, and a software exception handler is activated.

By setting the comparable flag in the AMD 80C287 math coprocessor status word equal to 1, the AMD 80C287 math coprocessor can indicate an exception. The AMD 80C287 math coprocessor, by checking the comparable exception mask in the control word, can determine if it should field the exception (mask = 1), or if it should indicate the exception to the CPU so that a software exception handler (mask = 0) can be activated.

When the exception is masked from user software (the mask is set), the AMD 80C287 math coprocessor returns its on-chip masked response for that exception. When the exception is unmasked (the mask is not set, that is, mask = 0), the AMD 80C287 math coprocessor returns its unmasked response. A masked response will generate a standard result and then continue the instruction. The unmasked response, by trapping to a software exception handler, lets the CPU acknowledge and act on the exception. All exception conditions and the AMD 80C287 math coprocessor's masked response are listed in Table 12-16 with a full description.

Note: The AMD 80C287 math coprocessor may find several exceptions in one instruction when exceptions are masked, because it keeps performing the instruction even after it has executed its masked response. The AMD 80C287 math coprocessor, for instance, could find a denormalized operand, execute its masked response to such an exception, and then find an underflow.

Table 12-16 Exception Conditions and Masked Responses

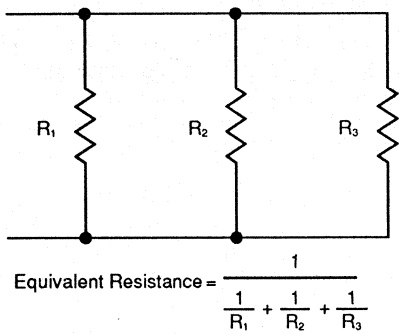
Condition	Masked Response
Invalid Operation	
Source register is tagged empty (usually due to stack underflow).	Return real indefinite.
Destination register is not tagged empty (usually due to stack overflow).	Return real indefinite (overwrite destination value).
One or both operands is a NaN.	Return NaN with larger absolute value (ignore signs).
(Compare and test operations only): one or both operands is a NaN.	Set condition codes Not Comparable.
(Addition operations only): closure is affine and operands are opposite-signed infinities; or closure is projective and both operands are ∞ (signs immaterial).	Return real indefinite.
(Subtraction operations only): closure is affine and operands are like-signed infinities; or closure is projective and both operands are ∞(signs immaterial).	Return real indefinite.
(Multiplication operations only): ∞ * 0; or 0 * ∞.	Return real indefinite.
Division operations only): ∞ ÷ ∞; or 0 ÷ 0; or 0 P pseudo zero; or divisor is denormal or unnormal.	Return real indefinite.
(FPREM instruction only): modulus (divisor) is unnormal or denormal; or dividend is ∞.	Return real indefinite, set condition code = Complete Remainder.
(FSQRT instruction only): operand is nonzero and negative; or operand is denormal or unnormal; or closure is affine and operand is −∞; or closure is projective and operand is ∞.	Return real indefinite.
(Compare operations only): closure is projective and ∞ is being compared with 0, a normal, or ∞.	Set condition code = Not Comparable.
(FTST instruction only): closure is projective and operand is ∞.	Set condition code = Not Comparable.
(FIST, FISTP instructions only): source register is empty, a NaN, denormal, unnormal, ∞, or exceeds representable range of destination.	Store integer indefinite.
(FBSTP instruction only): source register is empty, a NaN, denormal, unnormal, ∞, or exceeds 18 decimal digits.	Stored packed decimal indefinite.
(FST, FSTP instructions only): destination is short or long real and source register is an unnormal with exponent in range.	Store real indefinite.
(FXCH instruction only): one or both registers is tagged empty.	Change empty register(s) to real indefinite and then perform exchange.
Denormalized Operand	
(FLD instruction only): source operand is denormal.	No special action; load as usual.
(Arithmetic operations only): one or both operands is denormal.	Convert (in a work area) the operand to the equivalent unnormal; normalize as much as possible, and proceed with operation.
(Compare and test operations only): one or both operands is denormal or unnormal (other than pseudo zero).	Convert (in a work area) any denormal to the equivalent unnormal; normalize as much as possible, and proceed with operation.

Automatic Exception Handling

As discussed in the section above, the AMD 80C287 math coprocessor automatically performs an internal fix-up (masked-exception) response upon detection of an exception condition whose comparable mask bit in the AMD 80C287 math coprocessor control word is set. A default fix-up activity exists for any exception condition the AMD 80C287 math coprocessor may come across. Designed to be safe, such masked-exception responses are normally allowable for the majority of numeric applications.

In order to see how even extreme exceptions can be managed safely and automatically with the AMD 80C287 math coprocessor's default exception responses, imagine a calculation of the parallel resistance of various values in which just the standard formula (see Figure 12-10) is used. If R1 is zero, the circuit resistance is zero. The AMD 80C287 math coprocessor generates the valid result with the divide-by-zero and precision exceptions masked.

Figure 12-10 **Arithmetic Example Using Infinity**



14554A-058

By masking or unmasking certain numeric exceptions in the AMD 80C287 math coprocessor control word, AMD 80C287 math coprocessor programmers can make the AMD 80C287 math coprocessor responsible for the majority of exceptions, and assign the most extreme exceptions to programmed exception handlers. Exception-handling software is not easy to write, and the AMD 80C287 math coprocessor's masked responses are designed to return the most reasonable result for every condition. For most applications, masking all exceptions except Invalid Operation will deliver sufficient results with minimal programming effort. An Invalid Operation exception typically signals a fatal program error that must be handled; this exception is usually unmasked.

The exception flags in the AMD 80C287 math coprocessor status word furnish a complete record of exceptions that have been reported since these flags were last cleared. These flags can be cleared one of three ways once they are set:

1. Executing the FCLEX (clear exceptions) instruction
2. Reinitializing the AMD 80C287 math coprocessor
3. Overwriting the flags with an FRSTOR or FLDENV instruction

Thus, a programmer can mask every exception (except invalid operation), perform a calculation, and then check the status word for any exceptions found during the calculation.

Software Exception Handling

The AMD 80C287 math coprocessor signals unmasked exceptions to the 80C286 CPU by using the $\overline{\text{ERROR}}$ status line between the two processors.

When the 80C286 CPU next comes across a WAIT or ESC instruction in its stream of instructions, the 80C286, spotting the working condition of the $\overline{\text{ERROR}}$ status line, will automatically trap to an exception response routine via the Processor Extension Error exception (Interrupt #16).

The systems software usually contains this exception response routine. The following characterize normal exception responses:

- Incrementing an exception counter for future presentation or printing.
- Printing or showing diagnostic information (the AMD 80C287 math coprocessor environment and registers).
- Terminating further execution.
- Creating an instruction that runs without exception by using the exception pointers, and then executing it.

Application programmers using 80C286 systems with systems software support for the AMD 80C287 math coprocessor should check their references for the applicable system response to AMD 80C287 math coprocessor exceptions. More detailed information for system programmers on writing software exception handlers is found in the System-Level Numeric Programming section later in this manual.

The AMD 80C287 math coprocessor and the 8087 differ in how they signal numeric exceptions to the CPU. Only the 8087 needs an interrupt controller (8259A) to interrupt the CPU. When upgrading 8087 software to an AMD 80C287 math coprocessor, programmers should be conscious of these distinctions and any effects they may have on numeric exception-handling software. (See Appendix B for a more detailed explanation of the differences between the AMD 80C287 math coprocessor and the 8087.)

CHAPTER 13

PROGRAMMING NUMERIC APPLICATIONS

A broad range of instructions and programming alternatives are available to programmers who are designing applications for the AMD 80C287 math coprocessor.

The sections that follow give a detailed description of the AMD 80C287 instruction set, and proceed to a discussion on the various programming facilities.

THE AMD 80C287 INSTRUCTION SET

The operation of all AMD 80C287 instructions is discussed below. The instructions are organized in six functional classes within this section:

- Data Transfer instructions
- Arithmetic instructions
- Comparison instructions
- Transcendental instructions
- Constant instructions
- Processor Control instructions

This section ends by describing each instruction's execution speed, bus transfers, and exceptions. A coding example for each operand group the instruction will allow is also included. To simplify reference, this information is presented in a table that is alphabetical by instruction mnemonic.

This entire section describes the instruction set from the standpoint of the assembly programmer coding a program. The actual machine instruction encodings (mainly for reading unformatted memory dumps, controlling instruction retrieval on the bus, or writing exception handlers) are discussed in Appendix D.

Compatibility with the AMD 80C287 Math Coprocessor

The AMD 80C287 instruction set and the instruction set for the 8087, which is used with 8086 and 8088 systems, are practically identical. The majority of object programs created for the 8087 operate the same way on the AMD 80C287 math coprocessor. While some instructions are new to the AMD 80C287 math coprocessor, many 8087 instructions are identical for the AMD 80C287 math coprocessor. More information on these instruction set differences and on differences between the 8086 and 80C286 assemblers is included in Appendix E at the end of the manual.

Numeric Operands

The standard AMD 80C287 instruction can recognize one or two operands as inputs, operate on them, and return a result as an output. Register and memory locations (their contents) are most frequently operands, while the operands of some instructions are already determined. For example, the square root of the number in the top stack element is always found by the FSQRT instruction. In other instructions, however, the

programmer is allowed, or required, to explicitly code the operand(s) and the instruction mnemonic. Others allow one explicit operand and one implicit operand, typically the top stack element.

Sources and destinations are the two common operand types, whether they were provided by the programmer or used automatically. A source operand provides a single input to an instruction, which cannot change it. Even when an instruction changes the source operand's format (for example, real to integer), it occurs in an internal work area so as not to change the source operand. A destination operand can deliver an input to an instruction, as well, although it differs from a source operand, since its content can be changed when the operation returns its result. In other words, the result replaces the destination.

Several instructions let their operands be coded in different ways. FADD (add real), for example, may be written three ways: without operands, with only a source, or with a destination and a source. This section describes instructions by simply differentiating alternative operand forms with slashes, which are not coded. A choice of no explicit operands is denoted by double slashes. Thus, FADD's operands are described below:

```
//source/destination, source
```

That is, FADD can be written in one of the three ways listed below:

```
FADD
FADD source
FADD destination, source
```

Throughout this section, remember that any of the CPU's memory addressing modes can code memory operands. See Chapters 1–11 of this manual for a review of these modes (direct, register indirect, based, indexed, based indexed). Also see Table 13-17, which appears later in this chapter, for various examples of addressing mode.

Data Transfer Instructions

Operands are transferred among elements of the register stack, and between the stack top and memory, by these instructions (see Table 13-1 for a summary). Each data type (there are seven in all) can be transformed to temporary real and loaded (pushed) onto the stack in one operation, and similarly saved to memory. By automatically updating the AMD 80C287 tag word, the data transfer instructions can display the register contents after the instruction.

FLD SOURCE

By decrementing the stack pointer by one and then duplicating the source's content to the new stack top, FLD (load real) can load (push) the source operand onto the register stack's top. The source can be either a register on the stack (ST(i)) or one of the real data types in memory. Short and long real source operands automatically become temporary real. The stack top is copied by coding FLD ST(0).

FST DESTINATION

FST (store real) moves the stack top to the destination (either another stack register, or a short or long real memory operand). Provided the destination is short or long real, the significand rounds to the destination's width, according to the RC field of the control word; the exponent is transformed to the destination format's width and bias.

Table 13-1 Data Transfer Instructions

Real Transfers	
FLD	Load real
FST	Store real
FSTP	Store real and pop
FXCH	Exchange registers
Integer Transfers	
FILD	Integer load
FIST	Integer store
FISTP	Integer store and pop
Packed Decimal Transfers	
FBLD	Packed decimal (BCD) load
FBSTP	Packed decimal (BCD) store and pop

However, if the stack top is labeled special (it holds ∞, a NaN, or a denormal), then the stack top's significand is chopped (not rounded) on the right to fit the destination. Likewise, the exponent is not transformed, but chopped on the right and shifted as is. Thus, the value's identification is maintained as ∞ or a NaN (exponent all ones) or a denormal (exponent all zeros) so that it can be appropriately loaded and labeled later in the program.

FSTP DESTINATION

FSTP (store real and pop) works just like FST, but the stack is popped after the transfer. This is achieved by labeling the top stack element empty, and then incrementing ST. Unlike FST, FSTP allows saving to a temporary real memory variable. Coding FSTP ST(0) is the same as popping the stack without data transfer.

FXCH//DESTINATION

FXCH (exchange registers) switches the contents of the destination and the stack top registers. ST(1) is used when the destination is not explicitly coded. While several AMD 80C287 instructions execute only on the stack top, FXCH supplies an easy way to effectively utilize these instructions on lower stack elements. The sequence below, for instance, finds the square root of the third register from the top:

```
FXCH ST(3)
FSQRT
FXCH ST(3)
```

FILD SOURCE

FILD (integer load) transforms the source memory operand from its binary integer format (word, short, or long) to temporary real. The result is loaded (pushed) onto the stack. Provided every bit in the source was zero, the (new) stack top is labeled zero; in all other instances, it is labeled valid.

FIST DESTINATION

The content of the stack top is rounded by FIST (integer store) to an integer according to the RC field of the control word. The result is moved to the destination, which can define a word or short integer variable. Negative zero and positive zero are saved in one encoding: 0000...00.

FISTP DESTINATION

FISTP (integer and pop) works like FIST, and pops the stack after the move, as well. The destination can be any binary integer data type.

FBLD SOURCE

FBLD (packed decimal (BCD) load) transforms the content of the source operand from packed decimal to temporary real. The result is loaded (pushed) onto the stack. The source's sign, even where the value equals negative zero, is maintained. FBLD is a precise operation, as the source is loaded without rounding error.

The source's packed decimal digits are considered to be in the range 0–9H. The instruction does not, however, look for invalid digits (A–FH), and an attempt to load an invalid encoding returns an undefined result.

FBSTP//DESTINATION

FBSTP (packed decimal (BCD) store and pop) transforms the stack top's content to a packed decimal integer. The result is maintained at the destination in memory, and the stack is popped. By increasing the value by 0.5 and then chopping, this instruction can generate a rounded integer from a nonintegral value. Handle rounding by using FRNDINT before FBSTP.

Arithmetic Instructions

The AMD 80C287 arithmetic instruction set (Table 13-2) supplies extensive variations on the standard add, subtract, multiply, and divide operations, as well as many additional useful functions. Examples of these instructions include a simple absolute value to a square root instruction that performs quicker than normal division. Thus, AMD 80C287 programmers need not waste time eliminating square roots from algorithms because they don't work fast enough. Performing precise modulo division, rounding real numbers to integers, and scaling values by powers of two are among the other arithmetic instructions available.

The AMD 80C287 standard arithmetic instructions (addition, subtraction, multiplication, and division), which promote the development of extremely efficient algorithms, allow minimization of memory references and use of the AMD 80C287 register stack to its maximum capabilities.

The available operation/operand forms for basic arithmetic are listed in Table 13-3. Beyond the four standard operations, two reversed instructions exist to make subtraction and division symmetrical like addition and multiplication. The programmer has unconventional flexibility due to the many instruction and operand forms:

- Operands can be found in registers or memory.
- Results can be placed in a number of registers.
- Operands can be various AMD 80C287 data types: temporary real, long real, short real, short integer or word integer, with the AMD 80C287 math coprocessor automatically transforming to temporary real.

Table 13-2 Arithmetic Instructions

Addition	
FADD	Add real
FADDP	Add real and pop
FIADD	Integer add
Subtraction	
FSUB	Subtract real
FSUBP	Subtract real and pop
FISUB	Integer subtract
FSUBR	Subtract real reversed
FSUBRP	Subtract real reversed and pop
FISUBR	Integer subtract reversed
Multiplication	
FMUL	Multiply real
FMULP	Multiply real and pop
FIMUL	Integer multiply
Division	
FDIV	Divide real
FDIVP	Divide real and pop
FIDIV	Integer divide
FDIVR	Divide real reversed
FDIVRP	Divide real reversed and pop
FIDIVR	Integer divide reversed
Other Operations	
FSQRT	Square root
FSCALE	Scale
FPREM	Partial remainder
FRNDINT	Round to integer
FXTRACT	Extract exponent and significand
FABS	Absolute value
FCHS	Change sign

Table 13-3 indicates how the five general instruction forms can be utilized in all six operations. The classical stack form can make the AMD 80C287 math coprocessor run like a classical stack machine. Only the instruction mnemonic, and not operands, can be coded like this. The AMD 80C287 math coprocessor selects the source operand from the stack top, and picks the destination from the next stack element. In effect, the AMD 80C287 math coprocessor takes the result of popping the stack and performing the operation, returns it to the new stack top, and replaces the operand with it.

The register form is a standardized version of the classical stack form. That is, the programmer identifies the stack top as one operand and any register on the stack as the other one. Coding the stack top as the destination furnishes an easy method in which a constant, somewhere else in the stack, can be accessed from the stack top. For example, with converse coding, in which ST is the source operand, including the top in a register used as an accumulator is permitted.

Table 13-3 Basic Arithmetic Instruction and Operands

Classical stack	F*op*	{ ST(1),ST }	FADD	
Register	F*op*	ST(i),ST or ST,ST(i)	FSUB	ST,ST(3)
Register pop	F*op*P	ST(i),ST	FMULP	ST(2),ST
Real memory	F*op*	{ ST, } short-real/long-real	FDIV	AZIMUTH
Integer memory	FI*op*	{ ST, } word-integer/short-integer	FIDIV	N_PULSES

Braces({ }) enclose implied operands; those indicated below are not coded, and are listed simply for information.

op =			
	ADD	destination	← destination + source
	SUB	destination	← destination – source
	SUBR	destination	← source – destination
	MUL	destination	← destination · source
	DIV	destination	← destination ÷ source
	DIVR	destination	← source ÷ destination

Frequently, the stack top's operand is necessary for one operation, but is useless in the computation that follows. The register pop form can pick up the stack top as the source operand, and then throw it away by popping the stack. Coding operands of ST(1), ST with a register pop mnemonic resembles a classical stack operation in that the top is popped and the result stays at the new top.

The versatility of the AMD 80C287 arithmetic instructions is expanded by the two memory forms, which let a real number or a binary integer in memory be used directly as a source operand. This feature is extremely helpful in cases where operands are not used often enough for them to be held in registers.

Note: Any memory addressing mode can define these operands; they could be elements in arrays, structures, other data organizations, or simple scalars.

A discussion of the six basic operations follows Table 13-3, and the section proceeds with information on the seven arithmetic operations that remain.

Addition

```
FADD     //source/destination,source
FADDP    //destination/source
FIADD    source
```

The addition instructions (add real, add real and pop, integer add) take the sum of the source and destination operands and return the result to the destination. The stack top's operand may be doubled by coding:

```
FADD ST,ST(0)
```

Normal Subtraction

```
FSUB     //source/destination,source
FSUBP    //destination/source
FISUB    source
```

The usual subtraction instructions (subtract real,subtract real and pop, integer subtract) take the source operand and subtract it from the destination. They then return the result to the destination.

Reversed Subtraction

```
FSUBR    //source/destination,source
FSUBRP   //destination/source
FISUBR   source
```

The reversed subtraction instructions (subtract real reversed, subtract real reversed and pop, integer subtract reversed) take the destination and subtract it from the source, returning the result to the destination.

Multiplication

```
FMUL     //source/destination,source
FMULP    destination,source
FIMUL    source
```

The multiplication instructions (multiply real, multiply real and pop, integer multiply) take the product of the source and destination operands, and then return the result to the destination. The stack top's content is squared as a result of coding FMUL ST,ST(0).

Normal Division

```
FDIV     //source/destination,source
FDIVP    destination,source
FIDIV    source
```

The normal division instructions (divide real, divide real and pop, integer divide) take the destination, divide it by the source and then return the result to the destination.

Reversed Division

```
FDIVR    //source/destination, source
FDIVRP   destination,source
FIDIVR   source
```

The reversed division instructions (divide real reversed, divide real reversed and pop, integer divide reversed) take the source operand, divide it by the destination and then return the result to the destination.

FSQRT

FSQRT (square root) substitutes the top stack element's content with its square root. (Recall that the square root of –0 is defined as –0.)

FSCALE

FSCALE (scale) analyzes the value in ST(1) as an integer. It then takes the sum of this value and the exponent of the number in ST, which is equal to:

$ST \leftarrow ST \cdot 2^{ST(1)}$

In this manner, FSCALE can furnish fast multiplication or division by integral powers of 2. It is especially helpful when scaling a vector's elements.

Note: FSCALE presupposes that the scale factor in ST(1) is an integral value in the range $-2^{15} \leq X < 2^{15}$. If the value is not integral, but is within range and has a magnitude higher than 1, FSCALE utilizes the closest integer of less magnitude. In other words, the value is chopped toward 0. The instruction generates a result that is not defined and does not indicate an exception if the value is

beyond the range, or $0 < |X| < 1$. The suggested method is to load the scale factor from a word integer so accurate operation may be guaranteed.

FPREM

FPREM (partial remainder) executes modulo division of the top stack element by the next stack element (ST(1) is the modulus). FPREM generates a precise result, preventing a precision exception from occurring. The remainder's sign and the sign of the first dividend are identical.

The FPREM instruction executes consecutive scaled subtractions. Acquiring the precise remainder when the operands are very different in magnitude can waste extensive amounts of processing time. Since the AMD 80C287 math coprocessor cannot be disturbed during instructions, the remainder function could cause interrupt latency to seriously rise in these instances. Consequently, the instruction was created to be performed repeatedly within a loop managed by software.

In a single execution, FPREM can diminish a magnitude difference of up to 2^{64}. The function is finished if FPREM generates a remainder that is lower in value than the modulus. Bit C2 of the status word condition code is also cleared if this stipulation is met. C2 is set to 1 if the function remains to be finished, and ST becomes the partial remainder. By saving the status word after FPREM's execution, software is able to check C2. It keeps performing the instruction (with the partial remainder in ST as the dividend) until C2 is cleared. By comparing ST to ST(1), on the other hand, a program can establish if a function is finished. FPREM must be performed once more if ST > ST(1). The remainder is 0 if ST = ST(1), and the remainder is ST if ST < ST(1). If there is a more crucial interrupting routine needing the AMD 80C287 math coprocessor, it can invoke a context switch between the instructions in the remainder loop.

Decreasing arguments (operands) of periodic transcendental functions within the range these instructions allow is another significant use of FPREM. For example, the argument of the FPTAN (tangent) instruction must be less than $\pi/4$. Using $\pi/4$ as a modulus, FPTAN decreases an argument until it is within FPTAN's range. Since FPREM generates a precise result, roundoff error does not enter into the calculation from the argument reduction, even if multiple iterations are necessary to get the argument within range.

Note: The effect of a rounded period, not a rounded argument, results from the rounding of π.

The most insignificant three bits of the quotient produced by FPREM (in C_3, C_1, C_0) are also furnished by FPREM. Since FPREM finds the first angle in the correct one of eight $\pi/4$ segments of the unit circle (see Table 13-4), this is of significance in transcendental argument reduction, as well. C0 will equal the value of C3 prior to FPREM's execution, provided the quotient is under 4; if it is below 2, then C3 will equal the value of C1 prior to FPREM's execution.

FRNDINT

FRNDINT (round to integer) takes the top stack element and rounds it to an integer. Presume, for example, that within ST is the AMD 80C287 real number encoding of decimal value 155.625. Provided the RC field of the control word is set to down or chop, FRNDINT modifies the value to 155; if it is set to up or closest, the value is modified to 156.

FXTRACT

FXTRACT (extract exponent and significand) decomposes the number in the stack top into two numbers. These numbers are the equivalent of the actual value of the operand's exponent and significand fields. The first operand on the stack is now the exponent, and the significand is loaded onto the stack. Once FXTRACT is complete, ST (the new stack top) holds the value of the initial significand in the form of a real number: its sign and the operand's are identical, its exponent is 0 true (16,383 or 3FFFH biased), and its significand is the same as the initial operand's. ST(1) holds the value of the original operand's true (unbiased) exponent in the form of a real number. Provided the initial operand equals zero, FXTRACT generates zeros in ST and ST(1), and *both* are labeled as the initial operand.

To better grasp FXTRACT's operation, presume ST holds a number with a true exponent of +4 (i.e., 4003H is included in its exponent field). Following the execution of FXTRACT, ST(1) has the real number +4.0. Its sign is positive, its exponent field includes 4001H (+2 true), and its significand field is 1Δ00...00B. To sum up, the value in ST(1) will equal $1.0 \times 2^2 = 4$. Provided ST has an operand with a true exponent of –7 (i.e., 3FF8H is included in its exponent field), FXTRACT returns an exponent of –7.0. After executing the instruction, ST(1)'s sign and exponent fields include C001H (negative sign, true exponent of 2), and its significand is 1Δ1100...00B. So the value in ST(1) equals $-1.11 \times 2^2 = -7.0$. ST's sign and significand fields are identical to the initial operand's after FXTRACT is executed in the above instances, and its exponent field includes 3FFFH (0 true).

FXTRACT, along with FBSTP, helps transform numbers in AMD 80C287 temporary real format into decimal representations (for printing or presenting). FXTRACT also aids in debugging, since it lets the exponent and significant parts of a real number be analyzed individually.

FABS

FABS (absolute value) modifies the top stack element to its absolute value, giving it a positive sign.

FCHS

FCHS (change sign) changes the top stack element's sign.

Table 13-4 Condition Code Interpretation after FPREM

Condition Code				
C_3	C_2	C_1	C_0	Interpretation after FPREM
X	1	X	X	Incomplete Reduction; further iteration is required for complete reduction.
				Complete Reduction; C_1, C_3, and C_0 contain the three least-significant bits of quotient:
0	0	0	0	(Quotient) MOD 8 = 0
0	0	0	1	(Quotient) MOD 8 = 4
0	0	1	0	(Quotient) MOD 8 = 1
0	0	1	1	(Quotient) MOD 8 = 5
1	0	0	0	(Quotient) MOD 8 = 2
1	0	0	1	(Quotient) MOD 8 = 6
1	0	1	0	(Quotient) MOD 8 = 3
1	0	1	1	(Quotient) MOD 8 = 7

Comparison Instructions

Every instruction in Table 13-5 examines the top stack element, frequently in conjunction with another operand, and returns the result in the status word condition code. The standard operations are as follows: compare, test (compare with zero), and analyze (return label, sign, and normalization). In order to maximize algorithms by promoting direct comparisons with binary integers and real numbers in memory, and popping the stack following a comparison, particular forms of the compare operation are supplied.

The FSTSW (store status word) instruction can move the condition code to memory to be analyzed, after a comparison.

Note: Instructions outside of those in the comparison group are able to update the condition code. Save the status word right away after a comparison operation to guarantee that it is not changed accidentally.

FCOM //SOURCE

FCOM (compare real) takes the stack top and the source operand, and compares the two. The source operand can be either a register on the stack, or a short or long real memory operand. ST and ST(1) are compared if an operand is not coded. Positive and negative forms of zero compare equally as if they had no sign. Table 13-6 indicates how the condition codes show the order of the operands after an instruction.

NaNs and ∞ (projective) are incomparable; they return C3 = C0 = 1 (see Table13-6).

FCOMP //SOURCE

FCOMP (compare real and pop) functions the same as FCOM, and pops the stack, as well.

FCOMPP

FCOMPP (compare real and pop twice) functions the same as FCOM. In addition, it pops the stack twice, disposing of both operands. The stack top is compared to ST(1); operands cannot be explicitly coded.

FICOM SOURCE

FICOM (integer compare) transforms the source operand to temporary real, comparing it with the stack top. The operand can refer to a word or short binary integer variable.

FICOMP SOURCE

FICOMP (integer compare and pop) functions the same as FICOM, and also disposes of the value in ST by popping the stack.

FTST

FTST (test) compares the top stack element to zero when checking it. Table 13-7 illustrates the result, as posted to the condition codes.

FXAM

FXAM (examine) analyzes the top stack element's content as positive/negative and NaN/unnormal/denormal/normal/zero, or empty. Each condition code value produced by FXAM is explained in Table 13-8. An empty register may return four separate encodings, but both bits C3 and C0 of the condition code are 1 in each encoding. Ignore bits C2 and C1 when inspecting for empty.

Table 13-5 Comparison Instructions

FCOM	Compare real
FCOMP	Compare real and pop
FCOMPP	Compare real and pop twice
FICOM	Integer compare
FICOMP	Integer compare and pop
FTST	Test
FXAM	Examine

Table 13-6 Condition Code Interpretation after FCOM

Condition Code				
C3	C2	C1	C0	Interpretation after FCOM
0	0	X	0	ST > source
0	0	X	1	ST < source
1	0	X	0	ST = source
1	1	X	1	ST is not comparable

Table 13-7 Condition Code Interpretation after FTST

Condition Code				
C3	C2	C1	C0	Interpretation after FTST
0	0	X	0	ST > 0
0	0	X	1	ST < 0
1	0	X	0	ST = 0
1	1	X	1	ST is not comparable; (i.e., it is a NaN or projective infinity)

Table 13-8 FXAM Condition Code Settings

Condition Code				
C3	C2	C1	C0	Interpretation
0	0	0	0	+ Unnormal
0	0	0	1	+ NaN
0	0	1	0	– Unnormal
0	0	1	1	– NaN
0	1	0	0	+ Normal
0	1	0	1	+ ∞
0	1	1	0	– Normal
0	1	1	1	– ∞
1	0	0	0	+ 0
1	0	0	1	Empty
1	0	1	0	– 0
1	0	1	1	Empty
1	1	0	0	+ Denormal
1	1	0	1	Empty
1	1	1	0	– Denormal
1	1	1	1	Empty

Transcendental Instructions

The time-consuming core calculations for all normal trigonometric, inverse trigonometric, hyperbolic, inverse hyperbolic, logarithmic, and exponential functions are carried out by these instructions (see Table 13-9). Prologue and epilogue software can decrease arguments to the range the instructions permit, and structure the result so it corresponds with the initial arguments, if so required. The transcendentals, which run on the top one or two stack elements, report their results to the stack, as well.

Note: The transcendental instructions presume their operands are valid and within range. The instruction descriptions found within this section supply the permitted operand range of every instruction.

It is necessary to normalize all operands to a transcendental; denormals, unnormals, infinities, and NaNs are treated as invalid. (There are some functions that will allow zero operands and others that do not find them within range.) If a transcendental operand is invalid or beyond the range, the instruction generates an undefined result and does not report an exception. Prior to performing a transcendental, the programmer must guarantee that operands are valid and within range. FPREM can pull a valid operand within range in the case of periodic functions.

Table 13-9 Transcendental Instructions

FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	$2^x - 1$
FYL2X	$Y \cdot \log_2 X$
FYL2XP1	$Y \cdot \log_2(X + 1)$

FPTAN

$0 \leq ST(0) \leq \pi/4$

FPTAN (partial tangent) calculates the function $Y/X = TAN(\Theta)$, in which Θ is derived from the top stack element and must be within the range $0 \leq \Theta \leq \pi/4$. A ratio is the operation's end result. Y substitutes for Θ in the stack, and X is pushed, making it the new stack top.

The ratio result of FPTAN and the ratio argument of FPATAN promote the optimal calculation of the other trigonometric functions, such as SIN, COS, ARCSIN, and ARCCOS. These are taken from TAN and ARCTAN by basic trigonometric identities.

FPATAN

$0 \leq ST(1) \leq ST(0) < \infty$

FPATAN (partial arctangent) calculates the function $\Theta = ARCTAN(Y/X)$, in which X is derived from the top stack element and Y from ST(1). Y and X are required to recognize the inequality $0 \leq Y < X <$. The instruction pops the stack, replacing the Y operand on the (new) stack top with Θ.

F2XM1

$0 \le ST(0) \le 0.5$

F2XM1 (2 to the X minus 1) computes the function $Y = 2^X - 1$, in which X is derived from the stack top and must be within the range $0 \le X \le 0.5$. X is replaced by Y, the result, at the stack top.

The purpose of this instruction is to generate an extremely accurate result, even when X approaches 0. Increase the result of F2XM1 by one to generate $Y = 2^X$.

The formulas below indicate how values besides 2 can be raised to a power of X:

$$10^x = 2^{x \cdot LOG_2 10}$$
$$e^x = 2^{x \cdot LOG_2 e}$$
$$y^x = 2^{x \cdot LOG_2 Y}$$

The following section explains that the AMD 80C287 math coprocessor built-in instructions load constants $LOG_2 10$ and $LOG_2 e$, and the FYL2X instruction can compute $X \cdot LOG_2 Y$.

FYL2X

$0 < ST(0) < \infty \quad -\infty < ST(1) < \infty$

FYL2X (Y log base 2 of X) computes the function $Z = Y \cdot LOG_2 X$, in which X is derived from the stack top and Y from ST(1). The operands have to be be within the ranges $0 < X < \infty$ and $-\infty < Y < +\infty$. The instruction pops the stack and replaces the *Y* operand with *Z*, the result, at the (new) stack top.

Since a product is always necessary, this function maximizes log calculations to any base besides two:

$LOG_n 2 \cdot LOG_2 X$

FYL2XP1

$0 \le |ST(0)| < (1 - (\sqrt{2}/2))$
$-\infty < ST(1) < \infty$

FYL2XP1 (*Y* log base 2 of $(X + 1)$) computes the function $Z = Y \cdot LOG_2(X+1)$, in which *X* is derived from the stack top and has to be within the range $0 \le |X| < (1 - (\sqrt{2}/2))$. *Y*, derived from ST(1), has to be within the range $-\infty < Y < \infty$. FYL2XP1 pops the stack, and replaces *Y* with *Z*, the result, at the (new) stack top.

The accuracy of this instruction is greater than that of FYL2X when calculating the log of a number very close to 1, such as $1 + \varepsilon$, where $\varepsilon << 1$. Supplying ε instead of $1 + \varepsilon$ as the input to the function permits retention of more significant digits.

Constant Instructions

Table 13-10 lists each of these instructions, which load (push) a regularly-used constant onto the stack. The values, with complete temporary real precision (64 bits), have an accuracy of about 19 decimal digits. Since a temporary real constant takes up 10 memory bytes, the constant instructions, each two bytes in length, conserve storage and increase execution speed, as well as simplify programming.

FLDZ

FLDZ (load zero) loads (pushes) +0.0 onto the stack.

Table 13-10 Constant Instructions

FLDZ	Load +0.0
FLD1	Load +1.0
FLDPI	Load π
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

FLD1

FLD1 (load one) loads (pushes) +1.0 onto the stack.

FLDPI

FLDPI (load π) loads (pushes) π onto the stack.

FLDL2T

FLDL2T (load log base 2 of 10) loads (pushes) $LOG_2 10$ onto the stack.

FLDL2E

FLDL2E (load log base 2 of e) loads (pushes) $LOG_2 e$ onto the stack.

FLDLG2

FLDLG2 (load log base 10 of 2) loads (pushes) $LOG_{10} 2$ onto the stack.

FLDLN2

FLDLN2 (load log base e of 2) loads (pushes) $LOG_e 2$ onto the stack.

Processor Control Instructions

Table 13-11 contains processor control instructions not normally used in calculations. Instead, they supply control over the AMD 80C287 math coprocessor concerning system-level activities, such as initialization, exception handling, and task switching.

Several of the AMD 80C287 processor control instructions have two forms of assembler mnemonic, as listed in Table 13-11:

- A wait form that checks for unmasked numeric errors. FSTSW is an example of this form's mnemonic, prefixed only with an F.
- A no-wait form that disregards unmasked numeric errors. FNSTSW is an example of this form's mnemonic, prefixed with an FN.

When the control instruction is coded using the mnemonic's no-wait form, the assembler does not execute a wait instruction before the ESC instruction, nor does the CPU check the AMD 80C287 $\overline{\text{ERROR}}$ status line prior to executing the processor control instruction.

Just the processor-control instruction class has this extra no-wait form. The 80C286 automatically synchronizes each numeric instructions, as the CPU tests the $\overline{\text{BUSY}}$ status line and only performs the numeric instruction when this line is not activated. Therefore, a CPU wait instruction need not be executed before AMD 80C287 numeric instructions for the latter to execute properly.

Table 13-11 Processor Control Instructions

FINIT/FNINIT	Initialize processor
FSETPM	Set Protected Mode
FLDCW	Load control word
FSTCW/FNSTCW	Store control word
FSTSW/FNSTSW	Store status word
FSTSW AX/FNSTSW AX	Store status word to AX
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environment
FLDENV	Load environment
FSAVE/FNSAVE	Save state
FRSTOR	Restore state
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE	Free register
FNOP	No operation
FWAIT	CPU wait

Also note that no function in the AMD 80C287 math coprocessor is executed by the 8087 instructions FENI and FDISI. Should these opcodes be found in an 80C286 or AMD 80C287 instruction stream, the AMD 80C287 math coprocessor will execute no particular operation, nor will any internal states be affected. Programmers who would like to port numeric software from 8087 environments to the 80C286, though, should note that the chances of being able to completely port program sections that include these exception-handling instructions are slim. A more-detailed description of the differences between the AMD 80C287 math coprocessor and the 8087 is found in Appendix E.

FINIT/FNINIT

FINIT/FNINIT (initialize processor) inserts the AMD 80C287 math coprocessor in a recognizable state that is not affected by any earlier activity. This instruction's no-wait form forces the AMD 80C287 math coprocessor to terminate any earlier numeric operations that are presently executing in the NEU. This instruction executes a function comparable to a hardware RESET, but the present AMD 80C287 operating mode (either real address mode or protected mode) is unaffected by FINIT/FNINIT. FNINIT and FINIT differ in that only FINIT tests for unmasked numeric exceptions.

Note: If FNINIT performs during the execution of an earlier AMD 80C287 memory-referencing instruction, AMD 80C287 bus cycles in progress terminate. If the CPU finds a Coprocessor Segment Overrun Exception (Interrupt 9), this instruction may be required to clear the AMD 80C287 math coprocessor.

FSETPM

FSETPM (set protected mode) sets the AMD 80C287 operating mode to protected virtual address mode. When first initialized after hardware RESET, the AMD 80C287 math coprocessor runs in real address mode, just like the 80C286 CPU. However, once the AMD 80C287 math coprocessor enters protected mode, only a hardware RESET can switch the AMD 80C287 math coprocessor back to real address mode.

In protected mode, the AMD 80C287 exception pointers do not appear as they do in real address mode (see the FSAVE and FSTENV instructions below), which will be most obvious to people who write numeric exception handlers. The AMD 80C287 operating mode should not concern those who program basic applications.

FLDCW SOURCE

FLDCW (load control word) takes the word the source operand has specified, and makes it the present processor control word. This instruction generally determines or alters the AMD 80C287 operation mode. Note that the following is possible, as long as an exception bit in the status word is set: by loading a different control word that unmasks that exception and clears the interrupt enable mask, an instant interrupt request is produced prior to the next instruction's execution. The suggested method for switching modes is two-fold. Any exceptions should be cleared first, and then the new control word can be loaded.

FSTCW/FNSTCW DESTINATION

FSTCW/FNSTCW (store control word) saves the present processor control word to the memory location the destination has specified. Unlike FNSTCW, FSTCW tests for unmasked numeric exceptions.

FSTSW/FNSTSW DESTINATION

FSTSW/FNSTCW (store status word) saves the AMD 80C287 status word's present value to the destination operand in memory. This instruction can:

- Impose conditional branching after a comparison or FPREM instruction (FSTSW).
- Survey the AMD 80C287 math coprocessor to see if it is busy (FNSTSW).
- Activate exception handlers in environments not using interrupts (FSTSW).

Unlike FNSTSW, FSTSW tests for unmasked numeric exceptions.

FSTSW AX/FNSTSW AX

FSTSW AX/FNSTSW AX (store status word to AX), a special AMD 80C287 instruction, saves the AMD 80C287 status word's present value directly into the 80C286 AX register. Through this instruction, conditional branching is maximized in numeric programs, where the 80C286 CPU is required to check the status of multiple AMD 80C287 status bits. Unlike the non-waited form, the waited form tests for unmasked numeric exceptions.

This instruction, once executed, updates the 80C286 AX register with the AMD 80C287 status word prior to the CPU's execution of any additional instructions. Thus, the 80C286 is able to test the AMD 80C287 status word right away; no WAIT or other synchronization instructions are necessary.

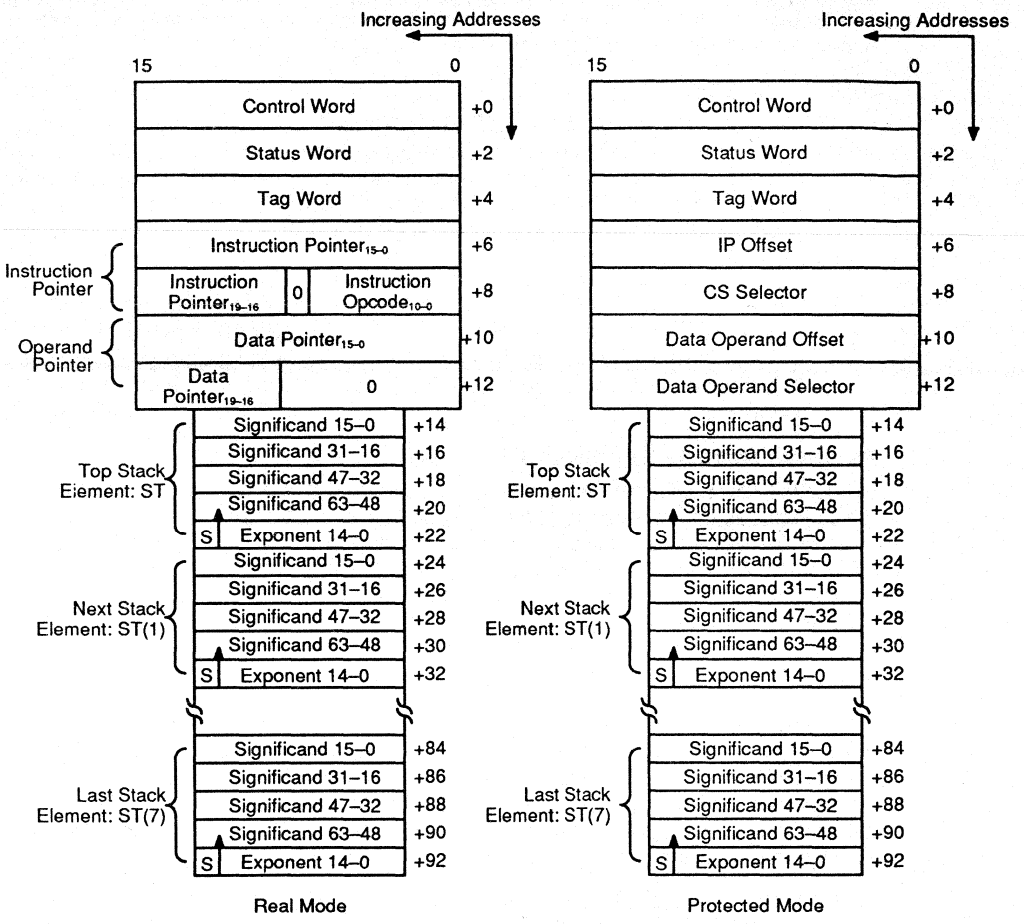
FCLEX/FNCLEX

FCLEX/FNCLEX (clear exceptions) clears every exception flag in the status word, as well as the error status flag and the busy flag. Consequently, the AMD 80C287 $\overline{\text{ERROR}}$ line is no longer active. Unlike FNCLEX, FCLEX tests for unmasked numeric exceptions.

FSAVE/FNSAVE DESTINATION

FSAVE/FNSAVE (save state) saves the entire AMD 80C287 state, which includes its environment and register stack, to the memory location the destination operand specifies. The 94-byte save area's layout is illustrated in Figure 13-1. The instruction is generally coded to store this image on the CPU stack. FNSAVE execution does not occur until all AMD 80C287 activity is regularly completed. In this way, after any running instruction is finished, the state of the AMD 80C287 math coprocessor is reflected by the save

Figure 13-1 FSAVE/FRSTOR Memory Layout



Notes:
S = Sign
Bit 0 of each field is rightmost, least significant bit or corresponding register field.
Bit 63 of significand is integer bit (assumed binary point is immediately to the right).

14554A-059

image. Once FSAVE/FNSAVE saves the state image to memory, it initializes the AMD 80C287 math coprocessor as if FINIT/FNINIT had performed.

The FSAVE/FNSAVE instruction is helpful whenever a program needs to store the present state of the AMD 80C287 math coprocessor and initialize it for a different routine. Below are three examples:

- An operating system must execute a context switch (suspend the running task to give a new task control).
- An exception handler must use the AMD 80C287 math coprocessor.
- An application task must pass a clean AMD 80C287 math coprocessor to a subroutine.

Unlike FNSAVE, FSAVE tests for unmasked numeric errors prior to execution. Before enabling CPU interrupts or performing any further AMD 80C287 instruction, an FWAIT should be performed. The execution of additional CPU instructions may occur between the FNSAVE/FSAVE and the FWAIT.

FRSTOR SOURCE

FRSTOR (restore state) loads the AMD 80C287 math coprocessor again from the 94-byte memory area the source operand specifies. An earlier FSAVE/FNSAVE instruction should have written this information, which should not have been modified by any other instruction. An FWAIT is not needed following FRSTOR, which will automatically wait and test for interrupts until every data transfer is finished before resuming with the next instruction.

Note: The AMD 80C287 math coprocessor reacts to its new state at the end of FRSTOR. For example, the AMD 80C287 math coprocessor will issue an exception request if the exception and mask bits in the memory image indicate this whenever the subsequent WAIT or error-checking-ESC instruction is performed.

FSTENV/FNSTENV DESTINATION

FSTENV/FNSTENV (store environment) saves the AMD 80C287 general status, which includes control, status, and tag words, and exception pointers, to the memory location the destination operand specified. The environment is usually stored on the CPU stack. Exception handlers frequently use FSTENV/FNSTENV because it furnishes access to the exception pointers that define the offending instruction and operand. FSTENV/FNSTENV sets every exception mask in the processor after it stores the environment. Unlike FNSTENV, FSTENV tests for pending errors prior to execution.

The layout of the environment data in memory is illustrated in Figure 13-2. All AMD 80C287 activity must be finished before FNSTENV stores the environment. Thus, the instruction stores the data, which reflects the AMD 80C287 math coprocessor following the execution of any earlier decoded instruction. Once FNSTENV/FSTENV saves the environment image to memory, it initializes the AMD 80C287 state as if FNINIT/FINIT had performed.

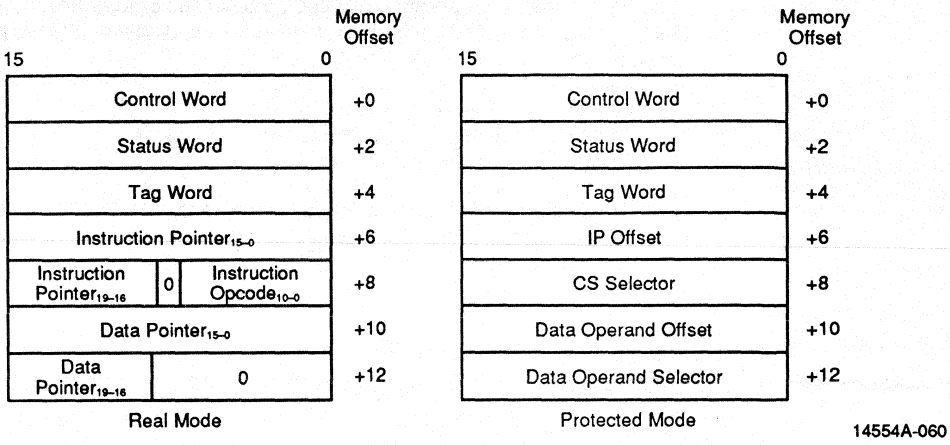
Until FSTENV/FNSTENV is permitted to finish, no other AMD 80C287 instruction can be decoded. An explicit FWAIT, or assembler-generated WAIT, should be completed before any further AMD 80C287 instruction, whenever FSTENV is coded.

FLDENV SOURCE

FLDENV (load environment) loads the environment again from the memory area the source operand specified. An earlier FSTENV/FNSTENV instruction should have written this data. CPU instructions (not concerning the environment image) may succeed FLDENV right away. An FWAIT is not needed following FLDENV, which will automatically wait for the completion of every data transfer prior to the next instruction's execution.

Note: Loading an environment image that includes an unmasked exception generates a numeric exception whenever the next WAIT or error-checking-ESC instruction is performed.

Figure 13-2 FSTENV/FLDENV Memory Layout



FINCSTP

FINCSTP (increment stack pointer) increases the stack top pointer (ST) in the status word by 1. It does not modify tags or register contents, or move data. Neither is it the same as popping the stack, since it does not set the earlier stack top's tag to empty. Incrementing the stack pointer when ST = 7 returns the result ST = 0.

FDECSTP

FDECSTP (decrement stack pointer) decreases ST, the stack top pointer in the status word, by 1. It does not modify tags or register, or move any data. The execution of FDECSTP when ST = 0 returns the result ST = 7.

FFREE DESTINATION

FFREE (free register) sets the destination register's tag to empty, while the register's content stays the same.

FNOP

FNOP (no operation) saves the stack top to the stack top (FST ST,ST(0)), executing nothing, in effect.

FWAIT (CPU INSTRUCTION)

FWAIT, not an actual AMD 80C287 instruction, is an optional mnemonic for the CPU WAIT instruction. Whenever the programmer synchronizes the CPU to the AMD 80C287 math coprocessor, he should code the FWAIT or WAIT mnemonic. In other words, he should delay additional instruction decoding until the AMD 80C287 math coprocessor finishes the active instruction. FWAIT tests for unmasked numeric exceptions.

Note: Only when the AMD 80C287 instruction is finished, a CPU instruction can try to access a memory operand. The coding below illustrates how to utilize FWAIT, forcing the CPU instruction to wait for the AMD 80C287 math coprocessor:

```
FIST    VALUE
FWAIT   ; Wait for FIST to complete
MOV     AX,VALUE
```

(See the section on Concurrent Processing with the AMD 80C287 Math Coprocessor in this chapter for additional information on when to code an FWAIT instruction.)

Instruction Set Reference Information

The operating characteristics of all the AMD 80C287 instructions appear later in this chapter in Table 13-13. A single table entry (the entries appear in alphabetical order for fast reference) is provided for every instruction mnemonic. Each entry lists the typical operand forms the instruction will permit and any exceptions that may occur in the operation.

Every group of operand types that can be coded with the mnemonic has a single entry. Refer to Table 13-12 for an explanation of the operand identifiers accepted in Table 13-13. Columns listing execution time in clocks, how many bus transfers run in the operation, the instruction's length in bytes, and an assembly coding example succeed this entry.

Table 13-12 Key to Operand Types

Identifier	Explanation
ST	Stack top; the register currently at the top of the stack.
ST(i)	A register in the stack i ($0 \leq i \leq 7$) stack elements from the top. ST(1) is the next-on-stack register, ST(2) is below ST(1), etc.
Short-real	A short real (32 bits) number in memory.
Long-real	A long real (64 bits) number in memory.
Temp-real	A temporary real (80 bits) number in memory.
Packed-decimal	A packed decimal integer (18 digits, 10 bytes) in memory.
Word-integer	A word binary integer (16 bits) in memory.
Short-integer	A short binary integer (32 bits) in memory.
Long-integer	A long binary integer (64 bits) in memory.
nn-bytes	A memory area *nn* bytes long.

INSTRUCTION EXECUTION TIME

Three main activities concern an AMD 80C287 instruction's execution. Each activity may add to the instruction's total execution time:

- 80C286 CPU overhead concerned with managing the ESC instruction opcode and setting up the AMD 80C287 math coprocessor.
- AMD 80C287 execution instruction.
- Operand transfers between the AMD 80C287 math coprocessor and memory or a CPU register.

Distinct clock frequencies of the 80C286 CPU and the AMD 80C287 math coprocessor affect the execution of the activities listed above. Other reasons why operand transfer times may increase include slow memories that necessitate the insertion of wait states in bus cycles, and bus contention because of additional processors in the system.

Analysts must consider every activity when computing a single numeric instruction's total execution time. Generally, it is presumed that the 80C286 has already obtained the numeric instructions that will be executed.

- Managing the ESC instruction opcode, the CPU overhead takes one CPU bus cycle before the AMD 80C287 math coprocessor starts to perform the numeric instruction. The CPU clock establishes this bus cycle's timing. The CPU must also set up the AMD 80C287 instruction and data pointer registers, but does not begin this activity until the AMD 80C287 math coprocessor starts instruction execution. Thus, overall execution time is unaffected by this concurrent activity.
- For every instruction, the length of execution time of separate numeric instructions on the AMD 80C287 math coprocessor differs. A normal execution clock count and a range for every AMD 80C287 instruction appears in Table 13-13. To convert an execution time into microseconds, divide the figures in the table by 10 (for a 10-MHz AMD 80C287 clock). Most often, this is an estimate of operand values that regularly describe the majority of applications. Best- and worst-case operand values, possible under extreme conditions, are included in the range.
- The operand transfer time needed to move operands between the AMD 80C287 math coprocessor and memory or a CPU register is dependent on the following: how many words are moved, the frequency of the CPU clock handling bus timing, how many wait states are added for slower memories, and whether operands are based at even or odd memory addresses. Because of the PEREQ/PEACK handshaking between the 80C286 and AMD 80C287 math coprocessor, and its asynchronous nature, some (very few) added bus cycles may be lost, as well. Depending on the frequencies of the CPU and AMD 80C287 clocks, this interaction can vary.

Table 13-13 illustrates the execution clock counts for the AMD 80C287 instruction execution, and presumes that, in the process of execution, no exceptions are found. Execution time is usually less than the normal figure, as a result of invalid operation, denormalized operand (unmasked), and zero divide exceptions. Execution, however, is still within the determined range. Execution time is unaffected by the precision exception. Table 13-14 indicates how unmasked overflow and underflow, and masked denormalized exceptions can implement further execution penalties. Thus, worst possible execution times are the sum of the high range figure and the greatest possible penalty.

Bus Transfers

Bus cycles that move operands between the AMD 80C287 math coprocessor and memory are necessary for AMD 80C287 instructions referencing memory. The operand's length and its alignment in memory determine the actual number of transfers. The first figure in Table 13-13 provides execution clocks for even-addressed operands; the second figure provides the clock count for odd-addressed operands.

Table 13-13 Instruction Set Reference Data

FABS

FABS (no operands)
Absolute value

Exceptions: I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	14	10-17	0	2	FABS

FADD

FADD //source/destination,source
Add real

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
//ST,ST(i)/ST(i),ST	85	70-100	0	2	FADD ST,ST(4)
short-real	105	90-120	2	2-4	FADD AIR_TEMP [SI]
long-real	110	95-125	4	2-4	FADD [BX].MEAN

FADDP

FADDP destination, source
Add real and pop

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
ST(i),ST	90	75-105	0	2	FADDP ST(2),ST

FBLD

FBLD source
Packed decimal (BCD) load

Exceptions: I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
packed-decimal	300	290-310	5	2-4	FBLD YTD_SALES

FBSTP

FBSTP destination
Packed decimal (BCD) store and pop

Exceptions: I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
packed-decimal	530	520-540	5	2-4	FBSTP [BX].FORECAST

Table 13-13 Instruction Set Reference Data, continued

FCHS

FCHS (no operands)
Change sign

Exceptions: I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	15	10-17	0	2	FCHS

FCLEX/FNCLEX

FCLEX/FNCLEX(no operands)
Clear exceptions

Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	5	2-8	0	2	FNCLEX

FCOM

FCOM //source
Compare real

Exceptions: I, D

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
//ST(i)	45	40-50	0	2	FCOM ST(1)
short-real	65	60-70	2	2-4	FCOM [BP].UPPER_LIMIT
long-real	70	65-75	4	2-4	FCOM WAVELENGTH

FCOMP

FCOMP //source
Compare real and pop

Exceptions: I, D

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
//ST(i)	47	42-52	0	2	FCOMP ST(2)
short-real	68	63-73	2	2-4	FCOMP [BP + 2].N_READINGS
long-real	72	67-77	4	2-4	FCOMP DENSITY

FCOMPP

FCOMPP (no operands)
Compare real and pop twice

Exceptions: I, D

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	50	45-55	0	2	FCOMPP

FDECSTP

FDECSTP (no operands)
Decrement stack pointer

Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	9	6-12	0	2	FDECSTP

FDIV

FDIV //source/destination,source
Divide real

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
//ST(i),ST	198	193-203	0	2	FDIV
short-real	220	215-225	2	2-4	FDIV DISTANCE
long-real	225	220-230	4	2-4	FDIV ARC [DI]

FDIVP

FDIVP destination, source
Divide real and pop

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
ST(i),ST	202	197-207	0	2	FDIVP ST(4),ST

FDIVR

FDIVR //source/destination, source
Divide real reversed

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
//ST,ST(i)/ST(i),ST	199	194-204	0	2	FDIVR ST(2),ST
short-real	221	216-226	2	2-4	FDIVR [BX].PULSE_RATE
long-real	226	221-231	4	2-4	FDIVR RECORDER.FREQUENCY

FDIVRP

FDIVRP destination, source
Divide real reversed and pop

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
ST(i),ST	203	198-208	0	2	FDIVRP ST(1),ST

Table 13-13 Instruction Set Reference Data, continued

FFREE — **FFREE** destination
Free register — **Exceptions:** None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
ST(i)	11	9-16	0	2	FFREE ST(1)

FIADD — **FIADD** source
Integer add — **Exceptions:** I, D, O, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
word-integer	120	102-137	1	2-4	FIADD DISTANCE_TRAVELLED
short-integer	125	108-143	2	2-4	FIADD PULSE_COUNT [SI]

FICOM — **FICOM** source
Integer compare — **Exceptions:** I, D

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
word-integer	80	72-86	1	2-4	FICOM TOOL.N_PASSES
short-integer	85	78-91	2	2-4	FICOM [BP+4].PARM_COUNT

FICOMP — **FICOMP** source
Integer compare and pop — **Exceptions:** I, D

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
word-integer	82	74-88	1	2-4	FICOMP [BP].LIMIT [SI]
short-integer	87	80-93	2	2-4	FICOMP N_SAMPLES

FIDIV — **FIDIV** source
Integer divide — **Exceptions:** I, D, Z, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
word-integer	230	224-238	1	2-4	FIDIV SURVEY.OBSERVATIONS
short-integer	236	230-243	2	2-4	FIDIV RELATIVE_ANGLE [DI]

Table 13-13 Instruction Set Reference Data, continued

FIDIVR **FIDIVR** source
Integer divide reversed
Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
word-integer	230	225-239	1	2-4	FIDIVR [BP].X_COORD
short-integer	237	231-245	2	2-4	FIDIVR FREQUENCY

FILD **FILD** source
Integer load
Exceptions: I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
word-integer	50	46-54	1	2-4	FILD [BX].SEQUENCE
short-integer	56	52-60	2	2-4	FILD STANDOFF [DI]
long-integer	64	60-68	4	2-4	FILD RESPONSE.COUNT

FIMUL **FIMUL** source
Integer multiply
Exceptions: I, D, O, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
word-integer	130	124-138	1	2-4	FIMUL BEARING
short-integer	136	130-144	2	2-4	FIMUL POSITION.Z_AXIS

FINCSTP **FINCSTP** (no operands)
Increment stack pointer
Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	9	6-12	0	2	FINCSTP

FINIT/FNINIT **FINIT/FNINIT** (no operands)
Initialize processor
Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	5	2-8	0	2	FINIT

Table 13-13 Instruction Set Reference Data, continued

FIST — **FIST** destination; Integer store — **Exceptions:** I, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
word-integer	86	80-90	1	2-4	FIST OBS.COUNT[SI]
short-integer	88	82-92	2	2-4	FIST [BP;].FACTORED_PULSES

FISTP — **FISTP** destination; Integer store and pop — **Exceptions:** I, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
word-integer	88	82-92	1	2-4	FISTP [BX].ALPHA_COUNT [SI]
short-integer	90	84-94	2	2-4	FISTP CORRECTED_TIME
long-integer	100	94-105	4	2-4	FISTP PANEL.N_READINGS

FISUB — **FISUB** source; Integer subtract — **Exceptions:** I, D, O, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
word-integer	120	102-137	1	2-4	FISUB BASE_FREQUENCY
short-integer	125	108-143	2	2-4	FISUB TRAIN_SIZE [DI]

FISUBR — **FISUBR** source; Integer subtract reversed — **Exceptions:** I, D, O, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
word-integer	120	103-139	1	2-4	FISUBR FLOOR [BX] [SI]
short-integer	125	109-144	2	2-4	FISUBR BALANCE

Table 13-13 Instruction Set Reference Data, continued

FLD

FLD source
Load real

Exceptions: I, D

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
ST(i)	20	17-22	0	2	FLD ST(0)
short-real	43	38-56	2	2-4	FLD READING [SI].PRESSURE
long-real	46	40-60	4	2-4	FLD [BP].TEMPERATURE
temp-real	57	53-65	5	2-4	FLD SAVEREADING

FLDCW

FLDCW source
Load control word

Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
2-bytes	10	7-14	1	2-4	FLDCW CONTROL_WORD

FLDENV

FLDENV source
Load environment

Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
14-bytes	40	35-45	7	2-4	FLDENV [BP + 6]

FLDLG2

FLDLG2 (no operands)
Load $\log_{10}2$

Exceptions: I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	21	18-24	0	2	FLDLG2

FLDLN2

FLDLN2 (no operands)
Load $\log_e2$

Exceptions: I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	20	17-23	0	2	FLDLN2

FLDL2E **FLDL2E** (no operands)
Load $\log_2 e$
Exceptions: I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	18	15-21	0	2	FLDL2E

FLDL2T **FLDL2T** (no operands)
Load $\log_2 10$
Exceptions: I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	19	16-22	0	2	FLDL2T

FLDPI **FLDPI** (no operands)
Load π
Exceptions: I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	19	16-22	0	2	FLDPI

FLDZ **FLDZ** (no operands)
Load +0.0
Exceptions: I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	14	11-17	0	2	FLDZ

FLD1 **FLD1** (no operands)
Load +1.0
Exceptions: I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	18	15-21	0	2	FLD1

Table 13-13 Instruction Set Reference Data, continued

FMUL

FMUL //source/destination,source
Multiply real

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
//ST(i),ST/ST,ST(i)[1]	97	90-105	0	2	FMUL ST,ST(3)
//ST(i),ST/ST,ST(i)	138	130-145	0	2	FMUL ST,ST(3)
short-real	118	110-125	2	2-4	FMUL SPEED_FACTOR
long-real[1]	120	112-126	4	2-4	FMUL [BP].HEIGHT
long-real	161	154-168	4	2-4	FMUL [BP].HEIGHT

FMULP

FMULP destination, source
Multiply real and pop

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
ST(i),ST[1]	100	94-108	0	2	FMULP ST(1),ST
ST(i),ST	142	134-148	0	2	FMULP ST(1),ST

FNOP

FNOP (no operands)
No operation

Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	13	10-16	0	2	FNOP

FPATAN

FPATAN (no operands)
Partial arctangent

Exceptions: U, P (operands not checked)

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	650	250-800	0	2	FPATAN

FPREM

FPREM (no operands)
Partial remainder

Exceptions: I, D, U

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	125	15-190	0	2	FPREM

Table 13-13 Instruction Set Reference Data, continued

FPTAN **FPTAN** (no operands)
Partial tangent

Exceptions: I, P (operands not checked)

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	450	30-540	0	2	FPTAN

FRNDINT **FRNDINT** (no operands)
Round to integer

Exceptions: I, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	45	16-50	0	2	FRNDINT

FRSTOR **FRSTOR** source
Restore saved state

Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
94-bytes		2	47	2-4	FRSTOR [BP]

FSAVE/FNSAVE **FSAVE/FNSAVE** destination
Save state

Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
94-bytes		3	47	2-4	FSAVE [BP]

FSCALE **FSCALE** (no operands)
Scale

Exceptions: I, O, U

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	35	32-38	0	2	FSCALE

Table 13-13 Instruction Set Reference Data, continued

FSETPM **FSETPM** (no operands)
Set protected mode
Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)		2-8	0	2	FSETPM

FSQRT **FSQRT** (no operands)
Square root
Exceptions: I, D, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	183	180-186	0	2	FSQRT

FST **FST** destination
Store real
Exceptions: I, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
ST(i)	18	15-22	0	2	FST ST(3)
short-real	87	84-90	2	2-4	FST CORRELATION [DI]
long-real	100	96-104	4	2-4	FST MEAN_READING

FSTCW/ FNSTCW **FSTCW** destination
Store control word
Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
2-bytes	15	12-18	1	2-4	FSTCW SAVE_CONTROL

FSTENV/ FNSTENV **FSTENV** destination
Store environment
Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
14-bytes	45	40-50	7	2-4	FSTENV [BP]

Table 13-13 Instruction Set Reference Data, continued

FSTP — **FSTP** destination
Store real and pop — **Exceptions:** I, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
ST(i)	20	17-24	0	2	FSTP ST(2)
short-real	89	86-92	2	2-4	FSTP [BX].ADJUSTED_RPM
long-real	102	98-106	4	2-4	FSTP TOTAL_DOSAGE
temp-real	55	52-58	5	2-4	FSTP REG_SAVE [SI]

FSTSW/ FNSTSW — **FSTSW** destination
Store status word — **Exceptions:** None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
2-bytes	15	12-18	1	2-4	FSTSW SAVE_STATUS

FSTSW AX/ FNSTSWAX — **FSTSW AX**
Store status word to AX — **Exceptions:** None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
AX		10-16	1	2	FSTSW AX

FSUB — **FSUB** //source/destination,source
Subtract real — **Exceptions:** I, D, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
//ST,ST(i)/ST(i),ST	85	70-100	0	2	FSUB ST,ST(2)
short-real	105	90-120	2	2-4	FSUB BASE_VALUE
long-real	110	95-125	4	2-4	FSUB COORDINATE.X

FSUBP — **FSUBP** destination, source
Subtract real and pop — **Exceptions:** I, D, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
ST(i),ST	90	75-105	0	2	FSUBP ST(2),ST

Table 13-13 Instruction Set Reference Data, continued

FSUBR

FSUBR //source/destination, source
Subtract real reversed

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
//ST,ST(i)/ST(i),ST	87	70-100	0	2	FSUBR ST,ST(1)
short-real	105	90-120	2	2-4	FSUBR VECTOR[SI]
long-real	110	95-125	4	2-4	FSUBR [BX].INDEX

FSUBRP

FSUBRP destination, source
Subtract real reversed and pop

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
ST(i),ST	90	75-105	0	2	FSUBRP ST(1),ST

FTST

FTST (no operands)
Test stack top against +0.0

Exceptions: I, D

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	42	38-48	0	2	FTST

FWAIT

FWAIT (no operands)
(CPU) Wait while AMD 80C287 coprocessor is busy

Exceptions: None (CPU instruction)

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	$3+5n^*$	$3+5n^4$	0	1	FWAIT

FXAM

FXAM (no operands)
Examine stack top

Exceptions: None

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	17	12-23	0	2	FXAM

Table 13-13 Instruction Set Reference Data, continued

FXCH — **FXCH** //destination
Exchange registers — **Exceptions:** I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
//ST(i)	12	10-15	0	2	FXCH ST(2)

FXTRACT — **FXTRACT** (no operands)
Extract exponent and significant — **Exceptions:** I

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	50	27-55	0	2	FXTRACT

FYL2X — **FYL2X** (no operands)
$Y \cdot Log_2X$ — **Exceptions:** P (operands not checked)

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	950	900-1100	0	2	FYL2X

FYL2XP1 — **FYL2XP1** (no operands)
$Y \cdot \log_2(X + 1)$ — **Exceptions:** P (operands not checked)

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	850	700-1000	0	2	FYL2XP1

F2XM1 — **F2XM1** (no operands)
$2^x - 1$ — **Exceptions:** U, P (operands not checked)

Operands	Execution Clocks		Operand Word Transfers	Code Bytes	Coding Example
	Typical	Range			
(no operands)	500	310-630	0	2	F2XM1

[1] Occurs when one or both operands is short—it has 40 trailing zeros in its fraction (e.g., it was loaded from a short-real memory operand).

[2] The AMD 80C287 execution clock count for this instruction is not meaningful in determining overall instruction execution time. For typical frequency ratios of the 80C286 and AMD 80C287 clocks, AMD 80C287 execution occurs in parallel with the operand transfers, with the operand transfers determining the overall execution time of the instruction. For 80C286:AMD 80C287 clock frequency ratios of 4:8, 1:1, and 8:5, the overall execution clock count for this instruction is estimated at 490-, 302-, and 227-80C287 clocks, respectively.

[3] The AMD 80C287 execution clock count for this instruction is not meaningful in determining overall instruction execution time. For typical frequency ratios of the 80C286 and AMD 80C287 clocks, AMD 80C287 execution occurs in parallel with the operand transfers, with the operand transfers determining the overall execution time of the instruction. For 80C286:80C287 clock frequency ratios of 4:8, 1:1, and 8:5, the overall execution clock count for this instruction is estimated at 376, 233, and 174 80C287 clocks, respectively.

[4] n = number of times CPU examines $\overline{\text{BUSY}}$ line before the AMD 80C287 completes execution of previous instruction.

Operands aligned at word boundaries (based at even memory addresses) mandate one bus cycle between the 80C286 data channel and memory, and one bus cycle to the AMD 80C287 math coprocessor, for each word transfer. Operands based at odd memory addresses mandate two bus cycles to move separate bytes between the 80C286 data channel and memory, and one bus cycle to the AMD 80C287 math coprocessor, for each word transfer.

Note: Align operands for the AMD 80C287 math coprocessor along word boundaries for optimal performance. In other words, base them at even memory addresses. Operands based at odd memory addresses move to memory effectively a byte at a time. They may take half again as long to move as operands aligned along word boundaries.

Table 13-14 Execution Penalties

Exception	Additional Clocks
Overflow (unmasked)	14
Underflow (unmasked)	16
Denormalized (masked)	33

When slow memories are used and wait states must be placed in the CPU bus cycle, extra transfer time is needed. The bus may not be available directly if there is more than one processor in the environment. Such overhead also contributes to a longer effective transfer time.

INSTRUCTION LENGTH

AMD 80C287 instructions are two bytes in length if they do not reference memory. Memory reference instructions fluctuate between two and four bytes. The 8- or 16-bit displacement values used along with the general 80C286 memory-addressing modes are located in the third and fourth bytes.

Note: The one-byte CPU wait instruction the assembler includes if the control instruction is coded by the wait form of the mnemonic (FINIT, FSTCW, FSTSW, FSTSW AX, FCLEX, FSTENV, and FSAVE) is not part of the lengths Table 13-13 cites for the processor control instructions (FNINIT, FNSTCW, FNSTSW, FNSTSW AX, FNCLEX, FNSTENV, and FNSAVE). The Processor Control Instructions section discussed wait and no-wait forms of the processor control instructions.

CONCURRENT PROCESSING WITH THE AMD 80C287 MATH COPROCESSOR

The 80C286 CPU and the AMD 80C287 math coprocessor can execute their respective instructions at the same time since they have their own execution units. Such simultaneous instruction execution is known as concurrency.

Special programming techniques are not necessary to benefit from concurrent execution; the AMD 80C287 numeric instructions are just put in line with the CPU's instructions. CPU and numeric instructions begin in the same order as the CPU recognizes them in its instruction stream. The CPU, however, can usually complete many of its instructions prior to the AMD 80C287 math coprocessor's completion of a numeric instruction activated beforehand, since numeric operations the AMD 80C287 math coprocessor executes usually need more time than operations the CPU performs.

Due to this concurrency, apparent benefits are available in terms of execution performance. A drawback to this concurrency is that numerous rules must be followed to guarantee valid synchronization of the 80C286 CPU and AMD 80C287 math coprocessor.

Concurrency in the AMD 80C287 math coprocessor is automatically handled and provided for by high-level languages. However, in return for the variability and performance of programming in assembly language, assembly-language programmers have to interpret and handle some aspects of concurrency. The assembly-language programmer or well-informed high-level-language programmer will find this section of particular use.

Managing Concurrency

Execution of the host and AMD 80C287 math coprocessor simultaneously is not difficult to create and support. Numeric programs have two major areas of activities: program control and arithmetic. Under program control, activities that determine what functions to execute, compute addresses of numeric operands, and control loops are performed. The arithmetic part of the activities performs simple operations on the numeric operands, among them addition, subtraction, and multiplication. The AMD 80C287 math coprocessor and host are equipped to manage these two areas individually and effectively.

Controlling concurrency is essential, since both the arithmetic and control areas must combine into a well-defined state, in which earlier arithmetic and control operations are finished and valid, before beginning further numeric operations.

Usually, the host lets the AMD 80C287 math coprocessor complete the present numeric operation before initiating another. This waiting period is known as synchronization.

Three kinds of synchronization are involved in controlling simultaneous execution of the AMD 80C287 math coprocessor:

1. Instruction synchronization
2. Data synchronization
3. Error synchronization

The appropriate compiler automatically furnishes all three synchronization types for higher-level language programmers. Assembly-language programmers are responsi-

ble for data and error synchronization; the AMD 80C287 interface ensures instruction synchronization.

Instruction Synchronization

The fact that the AMD 80C287 math coprocessor can execute only one numeric operation at a time necessitates instruction synchronization. Before the initiation of any numeric operation, the AMD 80C287 math coprocessor must have finished its prior instruction.

For the majority of ESC instructions, instruction synchronization is ensured, since the 80C286 automatically tests the $\overline{\text{BUSY}}$ status line from the AMD 80C287 math coprocessor before initiating execution of most ESC instructions. No explicit WAIT instructions are required to guarantee valid instruction synchronization.

Data Synchronization

Data synchronization concerns what happens when both the CPU and the AMD 80C287 math coprocessor reference the identical memory values within a certain block of code. These two processors are guaranteed by synchronization to access the memory operands in the correct order, just as a single processor without concurrency would access them. When the CPU and AMD 80C287 math coprocessor are using separate memory operands for a single numeric instruction, data synchronization is not important.

Below are two cases in which data synchronization might be of importance:

1. The 80C286 CPU reads or changes a memory operand first. It then causes the AMD 80C287 math coprocessor to load or change the identical operand.
2. The AMD 80C287 math coprocessor is induced to load or change a memory operand. Afterwards, the 80C286 CPU reads or changes the identical location.

Because of the AMD 80C287 interface instruction synchronization, data synchronization is automatically furnished for the first case. (The 80C286 always finishes its operation before it activates the AMD 80C287 math coprocessor.)

Data synchronization for the second case, however, may not always be automatic. Basically, no guarantee exists that the AMD 80C287 math coprocessor will complete its operation and access the memory operand before the 80C286 can reference the identical location.

Examples of the two possible instances of the CPU and AMD 80C287 math coprocessor with the same memory value are shown in Figure 13-3. In the first instance, the CPU will no longer need the operand before the AMD 80C287 math coprocessor accesses it, which is ensured by the AMD 80C287 math coprocessor. In the second instance, the CPU cannot reuse the memory operand until the AMD 80C287 math coprocessor is finished with it. FWAIT instructions, as these examples indicate, are necessary to guarantee this data synchronization.

Automatic data synchronization is supplied by many AMD 80C287 control instructions; yet, it is guarantee exists that the FSTSW/FNSTSW, FSTCW/FNSTCW, FLDCW, FRSTOR, and FLDENV instructions will complete their execution before the CPU reads or changes the referenced memory locations.

By issuing a request on the Coprocessor Data Channel before the CPU initiates its next instruction, the AMD 80C287 math coprocessor is able to supply data synchroni-

zation for these instructions. The CPU is not able to alter a memory value until the AMD 80C287 math coprocessor has been able to reference it, because the AMD 80C287 data transfers are processed before the CPU recovers control of the local bus. In the FSTSW AX instruction, for example, the 80C286 AX register is explicitly updated before the CPU resumes with the next instruction's execution.

In the case of the numeric instructions not discussed above, the assembly-language programmer should maintain an awareness of synchronization and be able to diagnose cases that require explicit data synchronization. Data synchronization can be supplied in two possible ways—either by programming an explicit FWAIT instruction, or by activating a later numeric instruction before accessing the operands or outcome of a prior instruction. Once the later numeric instruction's execution has been initiated, all memory references in previous numeric instructions are finished. Arriving at the subsequent host instruction following the synchronizing numeric instruction means that earlier numeric operands in memory can be accessed.

Figure 13-4 shows how to document the data-synchronization function of any FWAIT or numeric instruction properly. This documentation is necessary since a future alteration to the program could remove the synchronizing numeric instruction and induce program failure.

Data synchronization is automatically determined by high-level languages that handle it, but some applications in which a high-level language is unsuitable may exist.

Figure 13-3 Synchronizing References to Shared Data

```
; THE FOLLOWING ALL ALLOCATE THE CONSTANT: -126
; NOTE TWO'S COMPLETE STORAGE OF NEGATIVE BINARY INTEGERS.
;
; EVEN                                  ; FORCE WORD ALIGNMENT
WORD_INTEGER    DW   1111111110000010B ; BIT STRING
SHORT_INTEGER   DD   0FFFFFF82H        ; HEX STRING MUST START
                                       ; WITH DIGIT
LONG_INTEGER    DQ   -126              ; ORDINARY DECIMAL
SHORT_REAL      DD   -126.0            ; NOTE PRESENCE OF '.'
LONG_REAL       DD   -1.26E2           ; "SCIENTIFIC"
PACKED_DECIMAL DT   -126               ; ORDINARY DECIMAL INTEGER
; IN THE FOLLOWING, SIGN AND EXPONENT IS 'C005'
;     SIGNIFICAND IS '7E00...00', 'R' INFORMS ASSEMBLER THAT
;     THE STRING REPRESENTS A REAL DATA TYPE.
;
TEMP_REAL       DT   0C0057E00000000000000R  ; HEX STRING
```

Figure 13-4 Documenting Data Synchronization

```
; RESERVE SPACE FOR STATUS WORD
STATUS_WORD
; LAY OUT STATUS WORD FIELDS
STATUS RECORD
&     BUSY:                1,
&     COND_CODE3:          1,
&     STACK_TOP:           3,
&     COND_CODE2:          1,
&     COND_CODE1:          1,
&     COND_CODE0:          1,
&     INT_REQ:             1,
&     RESERVED:            1,
&     P_FLAG:              1,
&     U_FLAG:              1,
&     O_FLAG:              1,
&     Z_FLAG:              1,
&     D_FLAG:              1,
&     I_FLAG:              1
; POLL STATUS WORD UNTIL 80287 IS NOT BUSY
POLL:    FNSTSW   STATUS_WORD
         TEST     STATUS_WORD, MASK_BUSY
         JNZ      POLL
```

Figure 13-5 Nonconcurrent FIST Instruction Code Macro

```
SAMPLE      STRUC

   N_OBS             DD     ?   ; SHORT INTEGER
   MEAN              DQ     ?   ; LONG REAL
   MODE              DW     ?   ; WORD INTEGER
   STD_DEV           DQ     ?   ; LONG REAL
   ; ARRAY OF OBSERVATIONS -- WORD INTEGER
   TEST_SCORES       DW     1000 DUP (?)
SAMPLE ENDS
```

Automatic data synchronization can be acquired with the assembler, even though concurrency of execution is sacrificed as a result. The assembler can be modified to automatically place a WAIT instruction after the ESCAPE instruction so it may execute data synchronization. An example of how the assembly code macro can be modified for the FIST instruction to always place a WAIT instruction after the ESCAPE instruction is illustrated in Figure 13-5. The assembly source module contains this code macro. Any possible concurrency between the CPU and AMD 80C287 math coprocessor is sacrificed for this automatic data synchronization.

Error Synchronization

Just about any numeric instruction is capable of generating a numeric error under the wrong conditions. Synchronization for these errors, as for data references and numeric instructions, is required by simultaneous execution of the CPU and AMD 80C287 math coprocessor. Error synchronization, as a matter of fact, is automatically supplied by the synchronization that data and instructions require.

Nevertheless, the presence of inaccurate data or instruction synchronization may not become apparent until a numeric error has occurred. Furthermore, even though a

programmer does not anticipate that his numeric program will generate numeric errors, they may regularly occur in some systems. To better grasp these concepts, examine what can occur when the AMD 80C287 math coprocessor finds an error.

When a numeric exception happens, the AMD 80C287 math coprocessor can do one of the following:

- The AMD 80C287 math coprocessor can furnish a default fix-up for certain numeric errors. Programs can mask individual error types, specifying that the AMD 80C287 math coprocessor should produce a safe, acceptable result each time that error occurs. The AMD 80C287 math coprocessor regards the default error fix-up activity as within the instruction that is producing the error; the error is not identified externally. Although a flag is set in the numeric status register whenever errors are found, no information relaying where or when is available. Error synchronization will never be implemented if the AMD 80C287 math coprocessor invokes this default action for every error, but this is not grounds to ignore error synchronization.
- Another alternative to the AMD 80C287 default fix-up of numeric errors is informing the 80C286 CPU each time an exception occurs. Then the CPU may exercise any means of recovery procedures it wants, for any possible numeric error the AMD 80C287 math coprocessor could find. The AMD 80C287 math coprocessor halts execution of the numeric instruction and reports this activity to the CPU whenever a numeric error is unmasked and the error results. The CPU traps to a software exception handler whenever the next ESC or WAIT instruction occurs. The nonwaited forms of ESC instructions (FNINIT, FNSTENV, FNSAVE, FNSTSW, FNSTCW, and FNCLEX) do not test for the presence of errors.

The AMD 80C287 math coprocessor is asking for help whenever it reports an unmasked exception condition. Additional numeric program execution under the AMD 80C287 arithmetic and programming rules is unwarranted, since the error was unmasked.

As long as simultaneous execution is permitted, the CPU's state is not defined when it realizes the exception has occurred. It is possible that the CPU has modified several of its internal registers and is executing a completely separate program when the exception occurs. The AMD 80C287 math coprocessor manages this situation by updating special registers when each numeric instruction is initiated so they can describe the numeric program's state when the unsuccessful instruction was attempted.

Error synchronization guarantees that the AMD 80C287 math coprocessor is in a well-defined state once an unmasked numeric error has occurred. Without such a state, the exception recovery routines could not determine why the numeric error happened, or recover completely from the error.

Incorrect Error Synchronization

Figure 13-6 provides an example of how some instructions that are written without error synchronization are initially a success, but fail when shifted into a different environment.

Three instructions in Figure 13-6 load an integer, compute its square root, and then increment the integer. The AMD 80C287 interface and synchronous execution of the AMD 80C287 emulator let this program execute properly whenever there are no errors on the FILD instruction.

These circumstances change when the AMD 80C287 numeric register stack is extended to memory, which unmasks the invalid error. An invalid error occurs if either a push to a complete register or a pop from an empty register is made. The error's recovery routine must acknowledge the circumstances, clean up the stack, and then execute the initial operation.

In the first example of Figure 13-6, the recovery routine does not run properly since the value of COUNT is incremented before the AMD 80C287 math coprocessor is able to report the exception to the CPU. COUNT is incremented prior to the activation of the exception handler, and thus, the recovery routine loads an improper value of COUNT, inducing the program to not work or act erratically.

Proper Error Synchronization

Error Synchronization is dependent on the WAIT instructions that instruction and data synchronization, as well as the $\overline{\text{BUSY}}$ and $\overline{\text{ERROR}}$ signals of the AMD 80C287 math coprocessor, require. An unmasked error in the AMD 80C287 math coprocessor invokes the $\overline{\text{ERROR}}$ signal, notifying the CPU of a numeric error. The next time the CPU is aware of an error-checking ESC or WAIT instruction it will recognize the $\overline{\text{ERROR}}$ signal trapping automatically to Interrupt #16, the Coprocessor Error vector. Information essential to recover from the error will remain unaffected by the CPU if the ESC or WAIT instruction that follows is correctly placed.

Figure 13-6 Error Synchronization Examples

```
                    INCORRECT ERROR SYNCHRONIZATION
FILD    COUNT   ; NPX instruction
INC     COUNT   ; CPU instruction alters operand
FSQRT   COUNT   ; subsequent NPX instruction -- error from
                ;    previous NPX instruction detected here

                    PROPER ERROR SYNCHRONIZATION
FILD    COUNT   ; NPX instruction
FSQRT           ; subsequent NPX instruction -- error from
                ;    previous NPX instruction detected here
INC     COUNT   ; CPU instruction alters operand
```

CHAPTER 14

SYSTEM-LEVEL NUMERIC PROGRAMMING

A greater understanding of the AMD 80C287 math coprocessor is necessary for AMD 80C287 system programming than for application programming. The system programmer is responsible for emulation, initialization, exception handling, and data and error synchronization. A detailed discussion of these topics is covered in the sections below.

AMD 80C287 MATH COPROCESSOR ARCHITECTURE

While the AMD 80C287 math coprocessor seems like an extension of the 80C286 CPU on the software level, the mechanisms that let the 80C286 and AMD 80C287 math coprocessor interact are of a slightly higher complexity on the hardware level. This section covers the interaction of the AMD 80C287 math coprocessor and 80C286 CPU, pointing out the features that may interest systems programmers.

Processor Extension Data Channel

The internal Processor Extension Data Channel of the 80C286 executes all operand transfers between the AMD 80C287 math coprocessor and system memory. This autonomous, DMA-like data channel brings all the AMD 80C287 math coprocessor transfers of operands under the care of the 80C286 memory-management and protection mechanisms. On the software level, this data channel's operation is entirely transparent.

Extra bus drivers, controllers, or other components are not required to interface the AMD 80C287 math coprocessor to the local bus since the 80C286 executes all transfers between the AMD 80C287 math coprocessor and memory. The AMD 80C287 math coprocessor can access any memory available to the 80C286 CPU.

Real Address Mode and Protected Virtual Address Mode

Both the 80C286 CPU and the AMD 80C287 math coprocessor are operable in Real Address mode and in Protected mode. The AMD 80C287 math coprocessor defaults to Real Address mode after a hardware RESET. One privileged instruction (FSETPM) is required to initiate the AMD 80C287 math coprocessor in Protected mode.

Any memory location that is available to the task presently executing on the 80C286 is accessible to the AMD 80C287 math coprocessor. When the AMD 80C287 math coprocessor is in Protected mode, the 80C286's memory management and protection mechanisms automatically validate every memory reference, as well as all other memory references made by the task that is currently executing. Any protection violations affiliated with AMD 80C287 math coprocessor instructions will automatically induce the 80C286 to trap to a suitable exception handler.

These two AMD 80C287 operating modes appear different to the programmer only in how memory represents the AMD 80C287 instruction and data pointers after an FSAVE or FSTENV instruction. Memory represents the AMD 80C287 instruction and data pointers of the AMD 80C287 math coprocessor in Protected mode as a 16-bit

segment selector and a 16-bit offset. In Real Address mode, the identical instruction and data pointers of the AMD 80C287 math coprocessor are merely represented as the 20-bit physical addresses of the particular operands (see Figure 12-7).

Dedicated and Reserved I/O Locations

No memory addresses must be reserved for special purposes by the AMD 80C287 math coprocessor. The AMD 80C287 math coprocessor does, however, utilize I/O port addresses in the range 00F8H through 00FFH, even though these I/O operations are entirely transparent on the 80C286 software level. These reserved I/O addresses must not be directly referenced by 80C286 programs.

The 80C286's I/O Privilege Level (IOPL) should be utilized in reprogrammable environments with more than one user to limit application program access to the I/O address space to avoid any unintentional abuse or other misuse of numeric instructions in the AMD 80C287 math coprocessor. In this manner, the integrity of AMD 80C287 math coprocessor computations is ensured. (See Chapter 8 of the *80C286 Operating System Writer's Guide* for further information on the use of the I/O Privilege Level.)

PROCESSOR INITIALIZATION AND CONTROL

One of the systems software's main responsibilities concerns the activation, monitoring, and manipulation of the system hardware and software resources; this includes the AMD 80C287 math coprocessor. This section describes the following activities involving system initialization and control: acknowledging the AMD 80C287 math coprocessor, emulating the AMD 80C287 math coprocessor in software if the hardware is unavailable, and managing exceptions during the AMD 80C287 execution.

System Initialization

In the process of activating an 80C286 system, systems software is required to:

- Acknowledge if the AMD 80C287 math coprocessor is present or absent.
- Set flags in the 80C286 MSW to indicate the numeric environment's state.

If an AMD 80C287 math coprocessor is acknowledged, it must be:

- Activated.
- Set in Protected mode, if desired.

The above activities can be performed with speed and ease in the total system initialization.

Recognizing the AMD 80C287 Math Coprocessor

An example of a recognition routine that differentiates between the 80387 and the 8087 and decides if an AMD 80C287 math coprocessor is present is provided in Figure 14-1. Any 80386, 80C286, or 8086 hardware configuration with an AMD 80C287 math coprocessor socket can perform this routine.

This recognition routine, aware of the possibility of accidentally reading an expected value from a floating data bus when there is no AMD 80C287 math coprocessor, tries to prevent such an occurrence. Data read from a floating bus is not determined. While the routine anticipates reading a particular bit pattern from the AMD 80C287 math

Figure 14-1 Software Routine to Recognize the AMD 80C287 Math Coprocessor

```
8086/87/88/186 MACRO ASSEMBLER    Test for presence of a Numerics Chip, Revision 1.0                PAGE    1


DOS 3.20 (033-N) 8086/87/88/186 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE TEST_NPX
OBJECT MODULE PLACED IN FINDNPX.OBJ


LOC  OBJ                  LINE     SOURCE

                             1 +1  $title('Test for presence of a Numerics Chip, Revision 1.0')
                             2
                             3             name    Test_NPX
                             4
----                         5     stack   segment stack 'stack'
0000 (100                    6             dw      100 dup (?)
     ????
     )
00C8 ????                    7     sst     dw              ?
----                         8     stack   ends
                             9
----                        10     data    segment public 'data'
0000 0000                   11     temp    dw      0h
----                        12     data    ends
                            13
                            14     dgroup  group   data, stack
                            15     cgroup  group   code
                            16
----                        17     code    segment public 'code'
                            18             assume  cs:cgroup, ds:dgroup
                            19
0000                        20     start:
                            21     ;
                            22     ;       Look for an 8087, 80287, or 80387 NPX.
                            23     ;       Note that we cannot execute WAIT on 8086/88 if no 8087 is present.
                            24     ;
0000                        25     test_npx:
0000 90DBE3                 26             fninit                  ; Must use non-wait form
0003 BE0000          R      27             mov     si,offset dgroup:temp
0006 C7045A5A               28             mov     word ptr [si],5A5AH ; Initialize temp to non-zero value
000A 90DD3C                 29             fnstsw  [si]            ; Must use non-wait form of fstsw
                            30                                     ; It is not necessary to use a WAIT instruction
                            31                                     ;  after fnstsw or fnstcw.  Do not use one here.
000D 803C00                 32             cmp     byte ptr [si],0 ; See if correct status with zeroes was read
0010 752A                   33             jne     no_npx          ; Jump if not a valid status word, meaning no NPX
                            34     ;
                            35     ;       Now see if ones can be correctly written from the control word.
                            36     ;
0012 90D93C                 37             fnstcw  [si]            ; Look at the control word; do not use WAIT form
                            38                                     ; Do not use a WAIT instruction here!
0015 8B04                   39             mov     ax,[si]         ; See if ones can be written by NPX
0017 253F10                 40             and     ax,103fh        ; See if selected parts of control word look OK
001A 3D3F00                 41             cmp     ax,3fh          ; Check that ones and zeroes were correctly read
001D 751D                   42             jne     no_npx          ; Jump if no NPX is installed
                            43     ;
                            44     ;       Some numerics chip is installed.  NPX instructions and WAIT are now safe.
                            45     ;       See if the NPX is an 8087, 80287, or 80387.
                            46     ;       This code is necessary if a denormal exception handler is used or the
                            47     ;       new 80387 instructions will be used.
                            48     ;
001F 9BD9E8                 49             fld1                    ; Must use default control word from FNINIT
0022 9BD9EE                 50             fldz                    ; Form infinity
0025 9BDEF9                 51             fdiv                    ; 8087/287 says +inf = -inf
0028 9BD9C0                 52             fld     st              ; Form negative infinity
002B 9BD9E0                 53             fchs                    ; 80387 says +inf <> -inf
002E 9BDED9                 54             fcompp                  ; See if they are the same and remove them
0031 9BDD3C                 55             fstsw   [si]            ; Look at status from FCOMPP
0034 8B04                   56             mov     ax,[si]
0036 9E                     57             sahf                    ; See if the infinities matched
0037 7406                   58             je      found_87_287    ; Jump if 8087/287 is present
                            59     ;
```

Figure 14-1 Software Routine to Recognize the AMD 80C287 Math Coprocessor (continued)

```
8086/87/88/186 MACRO ASSEMBLER    Test for presence of a Numerics Chip, Revision 1.0

LOC  OBJ                  LINE     SOURCE

                            60     ;       An 80387 is present.  If denormal exceptions are used for an 8087/287,
                            61     ;       they must be masked.  The 80387 will automatically normalize denormal
                            62     ;       operands faster than an exception handler can.
                            63     ;
0039 EB0790                 64             jmp     found_387
003C                        65     no_npx:
                            66     ;       set up for no NPX
                            67     ;               ...
                            68     ;
003C EB0490                 69             jmp exit
003F                        70     found_87_287:
                            71     ;       set up for 87/287
                            72     ;               ...
                            73     ;
003F EB0190                 74             jmp exit
0042                        75     found_387:
                            76     ;       set up for 387
                            77     ;               ...
                            78     ;
0042                        79     exit:
....                        80     code    ends
                            81             end     start,ds:dgroup,ss:dgroup:sst

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

coprocessor, it defends itself against the bus' undefined state. The example also manages to be independent of any values in reserved bits. In this manner, it remains compatible with the later numerics coprocessors.

Configuring the Numerics Environment

After the 80C286 CPU acknowledges that the AMD 80C287 math coprocessor is present or absent, the 80C286 is required to place either the MP or the EM bit in its own machine status word as appropriate. The initialization routine has two options, listed below:

- Place the MP bit in the 80C286 MSW so numeric instructions can be performed directly by the AMD 80C287 component
- Place the EM bit in the 80C286 MSW, thereby allowing software emulation of the AMD 80C287 numeric instructions

The CPU is informed if an AMD 80C287 math coprocessor is physically accessible in the system by the Math Present (MP) flag of the 80C286 machine status word. The MP flag handles the WAIT instruction's function. During a WAIT instruction's execution, the 80C286 only checks the Task Switched (TS) bit if MP is set; the CPU traps to exception 7 if it detects the TS set under these circumstances.

The CPU is informed if AMD 80C287 functions are to be emulated by the Emulation Mode (EM) bit of the 80C286 machine status word. By automatically trapping program control to Exception #7 if the CPU learns EM is set while executing an ESC instruction, the exception handler is able to emulate the AMD 80C287 functions. Only with the LMSW (load machine status word) instruction (legal only at privilege level 0) can the 80C286 EM flag be modified; it can be analyzed using the SMSW (store machine status word) instruction (legal at any privilege level).

The WAIT instruction's function is managed by the EM bit, as well. The CPU does not test the ERROR pin for any error indication if the CPU learns EM is set while a WAIT is being executed.

Never set the EM bit simultaneously with MP. Either the MP or EM bit must be set to execute ESC instructions, but not both. See the AMD 80C287 Emulation section that appears later in this chapter for more details on software emulation.

Initializing the AMD 80C287 Math Coprocessor

Initializing the AMD 80C287 math coprocessor implies setting the AMD 80C287 math coprocessor in a recognizable state that is not affected by any activity executed previously. The example software routine in Table 14-1 used one FNINIT instruction to accept the AMD 80C287 math coprocessor in performing this initialization. As a result of this instruction, the AMD 80C287 math coprocessor is activated just as it is by the hardware RESET signal to the AMD 80C287 math coprocessor. Every error mask is set, and every register is labeled empty. The ST is set to zero, and the controls for default rounding, precision, and infinity are set as well. The state of the AMD 80C287 math coprocessor after initialization is illustrated in Table 14-1.

The AMD 80C287 math coprocessor is activated in Real Address mode after a hardware RESET signal like that following initial power-up. After the AMD 80C287 math coprocessor switches its operation to Protected mode (using the FSETPM instruction), only another hardware RESET can revert the processor to Real Address mode. The AMD 80C287 operating status is not changed by the FNINIT instruction.

AMD 80C287 Emulation

Systems software may decide upon the emulation of ESC instructions in software if no AMD 80C287 math coprocessor is available. The 80C286 hardware can support

Table 14-1 **AMD 80C287 Processor State Following Initialization**

Field	Value	Interpretation
Control Word		
Infinity Control	0	Projective
Rounding Control	00	Round to nearest
Precision Control	11	64 bits
Interrupt-Enable Mask	1	Interrupts disabled
Exception Mask	111111	All exceptions masked
Status Word		
Busy	0	Not busy
Condition Code	????	(Indeterminate)
Stack Top	000	Empty stack
Interrupt Request	0	No interrupt
Exception Flags	000000	No exceptions
Tag Word		
Tags	11	Empty
Registers	N.C.	Not changed
Exception Pointers		
Instruction Code	N.C.	Not changed
Instruction Address	N.C.	Not changed
Operand Address	N.C.	Not changed

this emulation with ease since it can be set to trap to a software emulation routine upon contact with an ESC instruction in its instruction stream.

As discussed earlier, whenever the 80C286 CPU comes in contact with an ESC instruction, and its MP and EM status bits are properly set (MP = 0, EM = 1), the 80C286 automatically traps to Interrupt #7, the Processor Extension Not Available exception. The return link saved on the stack targets the ESC instruction's first byte; this includes any prefix byte(s). The exception handler, able to use this return link to analyze the ESC instruction, can begin to emulate the numeric instruction in software.

Once it returns from the exception handler, the emulator must step the return pointer so that execution can continue from the first instruction after the ESC instruction.

An application program can barely distinguish execution on an 80C286 system with AMD 80C287 emulation from execution on an AMD 80C287 system, aside from their varying execution speeds.

When employing emulation on an 80C286 system, consider the following:

- Numeric applications, in conjunction with the emulator, must be executed in execute-readable code segments when operating in protected address mode. It is not possible to emulate numeric software if it is executed in execute-only code segments, because the emulator must be able to analyze the specific numeric instruction that induced the emulation trap.
- The 80C286 can be set in emulation mode by privileged tasks only. Privileged instructions are needed to set the 80C286 in emulation mode; they are not generally available to an application.

Handling Numeric Processing Exceptions

The AMD 80C287 math coprocessor may periodically need help to recover from numeric processing errors after the AMD 80C287 math coprocessor is initialized and regular execution of applications begins. Writing software exception handlers for such exceptions is covered in this section. The numeric processing exceptions themselves were previously discussed in this manual.

As described earlier, the AMD 80C287 math coprocessor has two possible reactions to acknowledging a numeric exception:

- In the case of a masked exception, the AMD 80C287 math coprocessor automatically executes its own masked exception response, which corrects the exception condition according to explicit rules. It then resumes executing its instruction.
- In the case of an unmasked exception, the AMD 80C287 math coprocessor uses the $\overline{\text{ERROR}}$ status line between the two processors to report the exception to the 80C286 CPU. Whenever the 80C286 comes across an ESC or WAIT instruction in its instruction stream, the CPU examines the $\overline{\text{ERROR}}$ status line's condition. If the CPU determines $\overline{\text{ERROR}}$ is active, it automatically traps to Interrupt vector #16, the Processor Extension Error trap.

Interrupt vector #16 will usually point to a software exception handler. The exception handler, which resembles an 80C286 interrupt procedure, may or may not be included in systems software.

The CPU is responsible for two things when managing numeric errors:

- The CPU must ignore the numeric context whenever an error is found.
- The CPU must clear the error and attempt to recover from it.

Although programmers treat these responsibilities differently depending on the implementation, the majority of exception handlers include the following basic steps:

- Save the AMD 80C287 environment, which contains control, status, and tag words, operand and instruction pointers, as it appeared when the exception was detected.
- Clear the exception bits in the status word.
- Enable interrupts on the CPU.
- Identify the exception by checking the status and control words in the save environment.
- Take action that is reliant on the system to correct the exception.
- Upon return to the interrupted program, commence regular execution.

The stored AMD 80C287 environment includes AMD 80C287 exception pointers, which take various forms determined by the AMD 80C287 mode of operation—Real Address mode or Protected mode. The manual previously discussed Real versus Protected mode, explaining how each operating mode presents this information.

Simultaneous Exception Response

If exceptions occur concurrently, the AMD 80C287 math coprocessor indicates one exception according to the precedence sequence in Table 14-2. Zero divided by zero, for example, results in an invalid operation instead of a zero divide exception.

Table 14-2 Precedence of AMD 80C287 Exceptions

Signaled First:	Denormalized operand (if unmasked) Invalid operation Zero Divide Denormalized (if masked) Over/Underflow
Signaled Last:	Precision

Exception Recovery Examples

Recovery routines for AMD 80C287 exceptions, able to assume various forms, can alter the AMD 80C287 arithmetic and programming rules. The effects of such changes may include a new definition of an error's default fix-up or the AMD 80C287 arithmetic, or a change in how the programmer views the AMD 80C287 math coprocessor.

A possible adjustment to an error response is to automatically normalize all denormals loaded from memory. Expanding the register stack into memory to supply an endless number of numeric registers changes appearance, while adjusting the AMD 80C287 arithmetic could automatically expand variables' precision and range when surpassed. Through numeric errors and related recovery routines, the AMD 80C287 math coprocessor can perform these functions so that they are transparent to the application programmer.

The following are other system-dependent actions that were discussed earlier:

- Incrementing an exception counter for future presentation or printing
- Printing or presenting diagnostic information (the AMD 80C287 environment and registers)
- Stopping additional execution
- Saving a diagnostic value (a NaN) in the result and resuming the computation

Note: Depending on the implementation, an exception does not necessarily constitute an error. The floating-point instruction that invoked the exception can resume after the exception handler fixes the error condition that signaled the exception. However, since the trap occurs at the ESC or WAIT instruction after the ESC instruction in question, the IRET instruction cannot perform this activity. The exception handler must first access the AMD 80C287 math coprocessor to get the offending instruction's address in the task that started it. Then the handler copies the address and performs the copy in the circumstance of the offending task. Afterwards, the handler returns to the present CPU instruction stream through the IRET instruction.

To fix the condition producing the numeric exception, exception handlers must be aware of the AMD 80C287 math coprocessor's exact state whenever the exception handler was activated, and be able to recreate it. It is essential that programmers understand precisely when, in the process of an AMD 80C287 instruction's execution, exceptions are signaled.

Prior to an operation's commencement, invalid operation, zero divide, and denormalized exceptions are located. Overflow, underflow, and precision exceptions, on the other hand, are detected only after a valid result has been computed. The unrevised AMD 80C287 register stack and memory act as if the offending instructions have not been performed each time a Before exception is raised.

The register stack and memory act as if the instruction is finished (that is, they may be revised) each time an After exception is detected. (In the case of a store or store-and-pop operation, however, unmasked over/underflow is treated like a before exception in that memory is unchanged and the stack is not popped.) Chapter 15 contains some programming examples that outline various exception handlers to manage numeric exceptions for the AMD 80C287 math coprocessor.

CHAPTER 15

NUMERIC PROGRAMMING EXAMPLES

The examples of AMD 80C287 numeric programs found in the sections below are meant to illustrate some available methods for programming the AMD 80C287 computing system for numeric applications.

CONDITIONAL BRANCHING EXAMPLES

Many numeric instructions inform the AMD 80C287 status word condition code bits of their results. Although conditional branching can be imposed several ways after a comparison, the standard approach is outlined below:

- Perform the comparison.
- Save the status word. (AMD 80C287 math coprocessor permits saving status directly into AX register.)
- Examine the condition code bits.
- Jump on the result.

A code fragment in Figure 15-1 shows how two memory-resident long real numbers could possibly be compared (such code could be utilized with the FTST instruction). The numbers labeled A and B are compared A to B.

The actual comparison must involve pushing A onto the top of the AMD 80C287 register stack and then comparing it to B, popping the stack with the identical instruction at the same time. Then the status word is saved to the 80286 AX register.

There are four possible orderings of A and B, bits C3, C2, and C0 of the condition code specify the particular ordering that holds. So the bits may correspond to the CPU's zero, parity, and carry flags (ZF, PF, and CF) each time the byte saves to the flags, they are placed in the upper byte of the AMD 80C287 status word. The code fragment positions ZF, PF, and CF of the CPU status word to correspond with the AMD 80C287 status word's values of C3, C2, and C0. Then, to check the flags, the fragment utilizes the CPU conditional jump instructions, producing a very compact code that requires just seven instructions.

The four condition code bits are updated by FXAM. Use of a jump table to establish the characteristics of the analyzed value is illustrated in Figure 15-2. The jump table (FXAM_TBL) is activated to include the 16-bit displacement of 16 labels, one for every condition code setting possible.

Note: Since four condition code settings are set to empty, an identical value appears in four of the table entries.

After executing FXAM and saving the status word, the program fragment adjusts the condition code bits, eventually generating a number in register BX equal to the condition code multiplied by 2. In this activity, the unused bits in the byte with the code are zeroed. This moves C3 to the right so that it is beside C2, and then shifts the code to multiply it by 2. The value generated becomes an index that chooses one of the displacements from FXAM_TBL (multiplying the condition code is necessary due to the

2-byte length of every value in FXAM_TBL). In effect, to process all results of the FXAM instruction, the unconditional JMP instruction vectors through the jump table to the indicated routine containing code (not present in the example).

Figure 15-1 Conditional Branching for Compares

```
            .
            .
            .
A           DQ          ?
B           DQ          ?
            .
            .
            .
            FLD         A           ; LOAD A ONTO TOP OF 287 STACK
            FCOMP       B           ; COMPARE A:B, POP A
            FSTSW       AX          ; STORE RESULT TO CPU AX REGISTER
            ;
            ; CPU AX REGISTER CONTAINS CONDITION CODES (RESULTS OF
            ; COMPARE)
            ;   LOAD CONDITION CODES INTO CPU FLAGS
            SAHF
            ;
            ; USE CONDITIONAL JUMPS TO DETERMINE ORDERING OF A TO
            ; B
            ;
            JP          A_B_UNORDERED   ; TEST C2 (PF)
            JB          A_LESS          ; TEST C0 (CF)
            JE          A_EQUAL         ; TEST C3 (ZF)
A_GREATER:                  ; C0 (CF) = 0, C3 (ZF) = 0
            .
            .
A_EQUAL:                    ; C0 (CF) = 0, C3 (ZF) = 1
            .
            .
A_LESS:                     ; C0 (CF) = 1, C3 (ZF) = 0
            .
            .
A_B_UNORDERED:              ; C2 (PF) = 1
            .
            .
```

Figure 15-2 Conditional Branching for FXAM

```
        ; JUMP TABLE FOR EXAMINE ROUTINE
        ;
FXAM_TBL  DW POS_UNNORM, POS_NAN, NEG_UNNORM, NEG_NAN,
&             POS_NORM, POS_INFINITY, NEG_NORM,
&             NEG_INFINITY, POS_ZERO, EMPTY, NEG_ZERO,
&             EMPTY, POS_DENORM, EMPTY, NEG_DENORM, EMPTY
        .
        .
        ;  EXAMINE ST AND STORE RESULT (CONDITION CODES)
        FXAM
        FSTSW AX

        ;
        ;  CALCULATE OFFSET INTO JUMP TABLE
        MOV     BH,0      ; CLEAR UPPER HALF OF BX,
        MOV     BL,AH     ; LOAD CONDITION CODE INTO BL
        AND     BL,00000111B      ; CLEAR ALL BITS EXCEPT C2-C0
        AND     AH,01000000B      ; CLEAR ALL BITS EXCEPT C3
        SHR     AH,2      ;  SHIFT C3 TWO PLACES RIGHT
        SAL     BX,1      ;  SHIFT C2-C0 1 PLACE LEFT (MULTIPLY
                          ;  BY 2)
        OR      BL,AH     ;  DROP C3 BACK IN ADJACENT TO C2
                          ;  (000XXXX0)
        ;
        ;  JUMP TO THE ROUTINE 'ADDRESSED' BY CONDITION CODE
        JMP     FXAM_TBL[BX]
        ;
        ;  HERE ARE THE JUMP TARGETS, ONE TO HANDLE
        ;    EACH POSSIBLE RESULT OF FXAM
POS_UNNORM:
        .
POS_NAN:
        .
NEG_UNNORM:
        .
NEG_NAN:
        .
POS_NORM:
        .
POS_INFINITY:
        .
NEG_NORM:
        .
NEG_INFINITY:
        .
POS_ZERO:
        .
EMPTY:
        .
NEG_ZERO:
        .
POS_DENORM:
        .
NEG_DENORM:
```

EXCEPTION HANDLING EXAMPLES

Exception handlers can be written several ways, but one helpful technique is considering the exception handler procedure as prologue, body, and epilogue sections of code. (Interrupt pointer (vector) #16 should initiate this procedure for compatibility with the AMD 80C287 emulators.)

CPU interrupts are disabled at the start of the prologue, which executes every function that must be shielded from any interruption by higher-priority sources. Normally, this means storing CPU registers and moving diagnostic information from the AMD 80C287 math coprocessor to memory. The prologue can allow CPU interrupts to let higher-priority interrupt handlers preempt the exception handler, once the necessary processing is finished.

After inspecting the diagnostic information, the exception handler body responds in a manner that is appropriately application-dependent. The body's response could stop execution, show a message, or attempt to fix the problem and resume regular execution.

Basically, the prologue's actions are reversed by the epilogue. That is, the CPU and the AMD 80C287 math coprocessor are restored so that regular execution may proceed. If the epilogue loads an unmasked exception flag into the AMD 80C287 math coprocessor, however, another exception will immediately occur.

The coding of three skeleton exception handlers is illustrated in Figures 15-3 through 15-5, which indicate how prologues and epilogues can be written for certain circumstances. However, the information in the figures shows only where the application-dependent exception handling body should be positioned. Figures 15-3 and 15-4 are alike, except for their choice of instructions to save and restore the AMD 80C287 math coprocessor. The sacrifice here is either the greater diagnostic information furnished by FNSAVE or the quicker execution of FNSTENV. For applications that are susceptible to interrupt latency or that are not required to inspect register contents, FNSTENV shortens the term of the critical region, during which the CPU will not acknowledge other interrupt requests unless they are nonmaskable.

The epilogues make the CPU and the AMD 80C287 math coprocessor ready to continue execution from where the interruption occurred (the instruction after the one that produced the unmasked exception).

Note: Before reloading, the exception flags in the memory image in the AMD 80C287 math coprocessor are set to zero. The whole status word image is cleared in these instances.

The examples shown in Figures 15-3 and 15-4 presume that the exception handler will not actually invoke an unmasked exception. In situations where this may occur, the basic method indicated in Figure 15-5 can be utilized. The general approach consists of storing the entire AMD 80C287 state and then loading a different control word in the prologue.

Note: When developing such an exception handler, be careful to prevent the handler from being continually reentered.

Figure 15-3 Full-State Exception Handler

```
SAVE_ALL          PROC
;
; SAVE CPU REGISTERS, ALLOCATE STACK SPACE
; FOR 80287 STATE IMAGE
     PUSH      BP
     MOV       BP,SP
     SUB       SP,94
; SAVE FULL 80287 STATE, WAIT FOR COMPLETION,
; ENABLE CPU INTERRUPTS
     FNSAVE    [BP-94]
     FWAIT
     STI
;
; APPLICATION-DEPENDENT EXCEPTION HANDLING
; CODE GOES HERE


;
; CLEAR EXCEPTION FLAGS IN STATUS WORD
; RESTORE MODIFIED STATE
; IMAGE
     MOV       BYTE PTR [BP-92], 0H
     FRSTOR    [BP-94]
; DE-ALLOCATE STACK SPACE, RESTORE CPU REGISTERS
     MOV       SP,BP
       .
       .
     POP       BP
;
; RETURN TO INTERRUPTED CALCULATION
     IRET
SAVE_ALL          ENDP
```

Figure 15-4 Reduced-Latency Exception Handler

```
SAVE_ENVIRONMENT PROC
;
; SAVE CPU REGISTERS, ALLOCATE STACK SPACE
; FOR 80287 ENVIRONMENT
      PUSH      BP

      MOV       BP,SP
      SUB       SP,14
; SAVE ENVIRONMENT, WAIT FOR COMPLETION,
; ENABLE CPU INTERRUPTS
      FNSTENV   [BP-14]
      FWAIT
      STI
;
; APPLICATION EXCEPTION-HANDLING CODE GOES HERE
;
; CLEAR EXCEPTION FLAGS IN STATUS WORD
; RESTORE MODIFIED
; ENVIRONMENT IMAGE
      MOV       BYTE PTR [BP-12], 0H
      FLDENV    [BP-14]
; DE-ALLOCATE STACK SPACE, RESTORE CPU REGISTERS
      MOV       SP,BP
      POP       BP
;
; RETURN TO INTERRUPTED CALCULATION
      IRET
SAVE_ENVIRONMENT ENDP
```

Figure 15-5 Reentrant Exception Handler

```
            .
            .
            .
       LOCAL_CONTROL   DW  ?   ; ASSUME INITIALIZED
            .
            .
            .
REENTRANT               PROC
;
; SAVE CPU REGISTERS, ALLOCATE STACK SPACE FOR
; 80287 STATE IMAGE
     PUSH       BP
            .
            .
            .
     MOV        BP,SP
     SUB        SP,94
; SAVE STATE, LOAD NEW CONTROL WORD,
; FOR COMPLETION, ENABLE CPU INTERRUPTS
     FNSAVE     [BP-94]
     FLDCW      LOCAL_CONTROL
     STI
            .
            .
            .
; APPLICATION EXCEPTION HANDLING CODE GOES HERE.
; AN UNMASKED EXCEPTION GENERATED HERE WILL
; CAUSE THE EXCEPTION HANDLER TO BE REENTERED.
; IF LOCAL STORAGE IS NEEDED, IT MUST BE
; ALLOCATED ON THE CPU STACK.
            .
            .
            .
; CLEAR EXCEPTION FLAGS IN STATUS WORD
; RESTORE MODIFIED STATE IMAGE
     MOV        BYTE PTR [BP-92], 0H
     FRSTOR     [BP-94]
; DE-ALLOCATE STACK SPACE, RESTORE CPU REGISTERS
     MOV        SP,BP
            .
            .
            .
     POP        BP
; RETURN TO POINT OF INTERRUPTION
     IRET
REENTRANT               ENDP
```

FLOATING-POINT TO ASCII CONVERSION EXAMPLES

Numeric programs are generally required to eventually format their results for the program user to display and inspect, usually as ASCII strings. Figure 15-6 explains the procedure for converting floating-point values to decimal ASCII character strings.

Instead of supplying the greatest number of significant digits possible, brevity, quickness, and validity were provided. To evade unneeded conversion errors, an attempt is made to retain integers within their particular domain.

A worst case accuracy of three units in the 16th decimal position for a noninteger value or integers greater than 10^{18} results from this routine (this is called double precision accuracy) whenever the extended precision real number format is applied. The accuracy is one unit in the 17th decimal position in the case of values with decimal exponents lower than a magnitude of 100.

Extra care in programming, increased program size, and reduced performance can result in greater precision.

Figure 15-6 Floating-Point to ASCII Conversion Routine

```
SERIES-III iAPX286 MACRO ASSEMBLER X108 ASSEMBLY OF MODULE FLOATING_TO_ASCII
OBJECT MODULE PLACED IN :F3:FPASC.OBJ
ASSEMBLER INVOKED BY:  ASM286 86 :F3:FPASC.AP2

LOC  OBJ                  LINE     SOURCE

                             1 +1  $title("80287 Floating-Point to 18-Digit ASCII Conversion")
                             2
                             3                     name    floating_to_ascii
                             4
                             5                     public  floating_to_ascii
                             6                     extrn   get_power_10:near, tos_status:near
                             7     ;
                             8     ;         This subroutine will convert the floating point number in the
                             9     ;       top of the 80287 stack to an ASCII string and separate power of 10
                            10     ;       scaling value (in binary).  The maximum width of the ASCII string
                            11     ;       formed is controlled by a parameter which must be > 1.  Unnormal values,
                            12     ;       denormal values, and psuedo zeroes will be correctly converted.
                            13     ;       A returned value will indicate how many binary bits of
                            14     ;       precision were lost in an unnormal or denormal value.  The magnitude
                            15     ;       (in terms of binary power) of a psuedo zero will also be indicated.
                            16     ;       Integers less than 10**18 in magnitude are accurately converted if the
                            17     ;       destination ASCII string field is wide enough to hold all the
                            18     ;       digits.  Otherwise the value is converted to scientific notation.
                            19     ;
                            20     ;         The status of the conversion is identified by the return value,
                            21     ;       it can be:
                            22     ;
                            23     ;               0       conversion complete, string_size is defined
                            24     ;               1       invalid arguments
                            25     ;               2       exact integer conversion, string_size is defined
                            26     ;               3       indefinite
                            27     ;               4       + NAN (Not A Number)
                            28     ;               5       - NAN
                            29     ;               6       + Infinity
                            30     ;               7       - Infinity
                            31     ;               8       psuedo zero found, string_size is defined
                            32     ;
                            33     ;         The PLM/286 calling convention is:
                            34     ;
                            35     ; floating_to_ascii:
                            36     ;       procedure (number,denormal_ptr,string_ptr,size_ptr,field_size,
                            37     ;               power_ptr) word external;
                            38     ;       declare (denormal_ptr,string_ptr,power_ptr,size_ptr) pointer;
                            39     ;       declare field_size word, string_size based size_ptr word;
                            40     ;       declare number real;
                            41     ;       declare denormal integer based denormal_ptr;
                            42     ;       declare power integer based power_ptr;
                            43     ;       end floating_to_ascii;
                            44     ;
                            45     ;         The floating point value is expected to be on the top of the NPX
                            46     ;       stack.  This subroutine expects 3 free entries on the NPX stack and
                            47     ;       will pop the passed value off when done.  The generated ASCII string
                            48     ;       will have a leading character either '-' or '+' indicating the sign
                            49     ;       of the value.  The ASCII decimal digits will immediately follow.
                            50     ;       The numeric value of the ASCII string is (ASCII_STRING)*10**POWER.
```

Figure 15-6 Floating-Point to ASCII Conversion Routine (continued)

```
LOC  OBJ                  LINE     SOURCE

                           51      ;       If the given number was zero, the ASCII string will contain a sign
                           52      ;       and a single zero chacter.  The value string_size indicates the total
                           53      ;       length of the ASCII string including the sign character.  String(0) will
                           54      ;       always hold the sign.  It is possible for string_size to be less than
                           55      ;       field_size.  This occurs for zeroes or integer values.  A psuedo zero
                           56      ;       will return a special return code.  The denormal count will indicate
                           57      ;       the power of two originally associated with the value.  The power of
                           58      ;       ten and ASCII string will be as if the value was an ordinary zero.
                           59      ;
                           60      ;         This subroutine is accurate up to a maximum of 18 decimal digits for
                           61      ;       integers.  Integer values will have a decimal power of zero associated
                           62      ;       with them.  For non integers, the result will be accurate to within 2
                           63      ;       decimal digits of the 16th decimal place (double precision).  The
                           64      ;       exponentiate instruction is also used for scaling the value into the
                           65      ;       range acceptable for the BCD data type.  The rounding mode in effect
                           66      ;       on entry to the subroutine is used for the conversion.
                           67      ;
                           68      ;         The following registers are not transparent.
                           69      ;
                           70      ;               ax bx cx dx si di flags
                           71      ;
                           72 +1   $eject
                           73      ;
                           74      ;               Define the stack layout.
                           75      ;
  0000[]                   76      bp_save         equ     word ptr [bp]
  0002[]                   77      es_save         equ     bp_save + size bp_save
  0004[]                   78      return_ptr      equ     es_save + size es_save
  0006[]                   79      power_ptr       equ     return_ptr + size return_ptr
  0008[]                   80      field_size      equ     power_ptr + size power_ptr
  000A[]                   81      size_ptr        equ     field_size + size field_size
  000C[]                   82      string_ptr      equ     size_ptr + size size_ptr
  000E[]                   83      denormal_ptr    equ     string_ptr + size string_ptr
                           84
                           85      parms_size      equ     size power_ptr + size field_size + size size_ptr +
  000A                     86      &                       size string_ptr + size denormal_ptr
                           87      ;
                           88      ;               Define constants used
                           89      ;
  0012                     90      BCD_DIGITS      equ     18              ; Number of digits in bcd_value
  0002                     91      WORD_SIZE       equ     2
  000A                     92      BCD_SIZE        equ     10
  0001                     93      MINUS           equ     1               ; Define return values
  0004                     94      NAN             equ     4               ; The exact values chosen here are
  0006                     95      INFINITY        equ     6               ; important.  They must correspond to
  0003                     96      INDEFINITE      equ     3               ; the possible return values and be in
  0008                     97      PSUEDO_ZERO     equ     8               ; the same numeric order as tested by
 -0002                     98      INVALID         equ     -2              ; the program.
 -0004                     99      ZERO            equ     -4
 -0006                    100      DENORMAL        equ     -6
 -0008                    101      UNNORMAL        equ     -8
  0000                    102      NORMAL          equ     0
  0002                    103      EXACT           equ     2
                          104      ;
                          105      ;               Define layout of temporary storage area.
                          106      ;
 -0002[]                  107      status          equ     word ptr [bp-WORD_SIZE]
 -0004[]                  108      power_two       equ     status - WORD_SIZE
 -0006[]                  109      power_ten       equ     power_two - WORD_SIZE
 -0010[]                  110      bcd_value       equ     tbyte ptr power_ten - BCD_SIZE
 -0010[]                  111      bcd_byte        equ     byte ptr bcd_value
 -0010[]                  112      fraction        equ     bcd_value
                          113
                          114      local_size      equ     size status + size power_two + size power_ten
  0010                    115      &                       + size bcd_value
                          116
 ----                     117      stack           stackseg (local_size+6) ; Allocate stack space for locals
                          118 +1   $eject
 ----                     119      code            segment er public
                          120                      extrn   power_table:qword
                          121      ;
                          122      ;               Constants used by this function.
                          123      ;
                          124                      even                    ; Optimize for 16 bits
0000 0A00                 125      const10         dw      10              ; Adjustment value for too big BCD
                          126      ;
                          127      ;               Convert the C3,C2,C1,C0 encoding from tos_status into meaningful bit
                          128      ;       flags and values.
                          129      ;
0002 F8                   130      status_table    db      UNNORMAL, NAN, UNNORMAL + MINUS, NAN + MINUS,
0003 04
0004 F9
0005 05
0006 00                   131      &                       NORMAL, INFINITY, NORMAL + MINUS, INFINITY + MINUS,
0007 06
0008 01
0009 07
000A FC                   132      &                       ZERO, INVALID, ZERO + MINUS, INVALID,
000B FE
000C FD
000D FE
000E FA                   133      &                       DENORMAL, INVALID, DENORMAL + MINUS, INVALID
000F FE
0010 FB
0011 FE
```

Figure 15-6 Floating-Point to ASCII Conversion Routine (continued)

```
LOC  OBJ                  LINE    SOURCE

                          134
0012                      135     floating_to_ascii proc
                          136
0012 E80000         E     137             call    tos_status              ; Look at status of ST(0)
0015 8BD8                 138             mov     bx,ax                   ; Get descriptor from table
0017 2E8A870200     R     139             mov     al,status_table[bx]
001C 3CFE                 140             cmp     al,INVALID              ; Look for empty ST(0)
001E 752B                 141             jne     not_empty
                          142     ;
                          143     ;         ST(0) is empty'  Return the status value
                          144     ;
0020 C20A00               145             ret     parms_size
                          146     ;
                          147     ;         Remove infinity from stack and exit
                          148     ;
0023                      149     found_infinity
                          150
0023 DDD8                 151             fstp    st(0)                   ; OK to leave fstp running
0025 EB02                 152             jmp     short exit_proc
                          153     ;
                          154     ;         String space is too small'  Return invalid code
                          155     ;
0027                      156     small_string
                          157
0027 B0FE                 158             mov     al,INVALID
                          159
0029                      160     exit_proc:
                          161
0029 C9                   162             leave                           ; Restore stack
002A 07                   163             pop     es
002B C20A00               164             ret     parms_size
                          165     ;
                          166     ;         ST(0) is NAN or indefinite.  Store the value in memory and look
                          167     ;       at the fraction field to separate indefinite from an ordinary NAN
                          168     ;
002E                      169     NAN_or_indefinite:
                          170
002E DB7EF0               171             fstp    fraction                ; Remove value from stack for examination
0031 A801                 172             test    al,MINUS                ; Look at sign bit
0033 9B                   173             fwait                           ; Insure store is done
0034 74F3                 174             jz      exit_proc               ; Can't be indefinite if positive
                          175
0036 BB00C0               176             mov     bx,0C000H               ; Match against upper 16 bits of fraction
0039 2B5EF6               177             sub     bx,word ptr fraction+6  ; Compare bits 63-48
003C 0B5EF4               178             or      bx,word ptr fraction+4  ; Bits 32-47 must be zero
003F 0B5EF2               179             or      bx,word ptr fraction+2  ; Bits 31-16 must be zero
0042 0B5EF0               180             or      bx,word ptr fraction    ; Bits 15-0 must be zero
0045 75E2                 181             jnz     exit_proc
                          182
0047 B003                 183             mov     al,INDEFINITE           ; Set return value for indefinite value
0049 EBDE                 184             jmp     exit_proc
                          185     ;
                          186     ;         Allocate stack space for local variables and establish parameter
                          187     ;       addressibility
                          188     ;
004B                      189     not_empty:
                          190
004B 06                   191             push    es                      ; Save working register
004C C8100000             192             enter   local_size,0            ; Format stack
                          193
0050 8B4E08               194             mov     cx,field_size           ; Check for enough string space
0053 83F902               195             cmp     cx,2
0056 7CCF                 196             jl      small_string
                          197
0058 49                   198             dec     cx                      ; Adjust for sign character
0059 83F912               199             cmp     cx,BCD_DIGITS           ; See if string is too large for BCD
005C 7603                 200             jbe     size_ok
                          201
005E B91200               202             mov     cx,BCD_DIGITS           ; Else set maximum string size
                          203
0061                      204     size_ok:
                          205
0061 3C06                 206             cmp     al,INFINITY             ; Look for infinity
0063 7DBE                 207             jge     found_infinity          ; Return status value for + or - inf.
                          208
0065 3C04                 209             cmp     al,NAN                  ; Look for NAN or INDEFINITE
0067 7DC5                 210             jge     NAN_or_indefinite
                          211     ;
                          212     ;         Set default return values and check that the number is normalized.
                          213     ;
0069 D9E1                 214             fabs                            ; Use positive value only
                          215                                             ; sign bit in al has true sign of value
006B 8BD0                 216             mov     dx,ax                   ; Save return value for later
006D 33C0                 217             xor     ax,ax                   ; Form 0 constant
006F 8B7E0E               218             mov     di,denormal_ptr         ; Zero denormal count
0072 8905                 219             mov     word ptr [di],ax
0074 8B5E06               220             mov     bx,power_ptr            ; Zero power of ten value
0077 8907                 221             mov     word ptr [bx],ax
0079 80FAFC               222             cmp     dl,ZERO                 ; Test for zero
007C 732B                 223             jae     real_zero               ; Skip power code if value is zero
                          224
007E 80FAFA               225             cmp     dl,DENORMAL             ; Look for a denormal value
0081 732C                 226             jae     found_denormal          ; Handle it specially
                          227
```

Figure 15-6 Floating-Point to ASCII Conversion Routine (continued)

```
LOC  OBJ                LINE    SOURCE

0083 D9F4                228             fxtract                         ; Separate exponent from significand
0085 80FAF8              229             cmp     dl,UNNORMAL             ; Test for unnormal value
0088 7240                230             jb      normal_value
                         231
008A 80EAF8              232             sub     dl,UNNORMAL-NORMAL      ; Return normal status with correct sign
                         233     ;
                         234     ;         Normalize the fraction, adjust the power of two in ST(1) and set
                         235     ;       the denormal count value
                         236     ;
                         237     ;       Assert:  0 <= ST(0) < 1.0
                         238     ;
008D D9E8                239             fld1                            ; Load constant to normalize fraction
                         240
008F                     241     normalize_fraction:
                         242
008F DCC1                243             fadd    st(1),st                ; Set integer bit in fraction
0091 DEE9                244             fsub                            ; Form normalized fraction in ST(0)
0093 D9F4                245             fxtract                         ; Power of two field will be negative
                         246                                             ; of denormal count
0095 D9C9                247             fxch                            ; Put denormal count in ST(0)
0097 DF15                248             fist    word ptr [di]           ; Put negative of denormal count in memory
0099 DEC2                249             faddp   st(2),st                ; Form correct power of two in st(1)
                         250                                             ; OK to use word ptr [di] now
009B F71D                251             neg     word ptr [di]           ; Form positive denormal count
009D 7528                252             jnz     not_psuedo_zero
                         253     ;
                         254     ;         A psuedo zero will appear as an unnormal number.  When attempting
                         255     ;       to normalize it, the resultant fraction field will be zero.  Performing
                         256     ;       an fxtract on zero will yield a zero exponent value
                         257     ;
009F D9C9                258             fxch                            ; Put power of two value in st(0)
00A1 DF1D                259             fistp   word ptr [di]           ; Set denormal count to power of two value
                         260                                             ; Word ptr [di] is not used by convert
                         261                                             ; integer, OK to leave running
00A3 80EAF8              262             sub     dl,NORMAL-PSUEDO_ZERO   ; Set return value saving the sign bit
00A6 E9A400              263             jmp     convert_integer         ; Put zero value into memory
                         264     ;
                         265     ;         The number is a real zero, set the return value and setup for
                         266     ;       conversion to BCD
                         267     ;
00A9                     268     real_zero:
                         269
00A9 80EAFC              270             sub     dl,ZERO-NORMAL          ; Convert status to normal value
00AC E99E00              271             jmp     convert_integer         ; Treat the zero as an integer
                         272     ;
                         273     ;         The number is a denormal.  FXTRACT will not work correctly in this
                         274     ;       case.  To correctly separate the exponent and fraction, add a fixed
                         275     ;       constant to the exponent to guarantee the result is not a denormal.
                         276     ;
00AF                     277     found_denormal:
                         278
00AF D9E8                279             fld1                            ; Prepare to bump exponent
00B1 D9C9                280             fxch
00B3 D9F8                281             fprem                           ; Force denormal to smallest representable
                         282                                             ; extended real format exponent
00B5 D9F4                283             fxtract                         ; This will work correctly now
                         284     ;
                         285     ;         The power of the original denormal value has been safely isolated.
                         286     ;       Check if the fraction value is an unnormal.
                         287     ;
00B7 D9E5                288             fxam                            ; See if the fraction is an unnormal
00B9 9BDFE0              289             fstsw   ax                      ; Save 80287 status in CPU AX reg for later
00BC D9C9                290             fxch                            ; Put exponent in ST(0)
00BE D9CA                291             fxch    st(2)                   ; Put 1.0 into ST(0), exponent in ST(2)
00C0 80EAFA              292             sub     dl,DENORMAL-NORMAL      ; Return normal status with correct sign
00C3 A90044              293             test    ax,4400H                ; See if C3=C2=0 impling unnormal or NAN
00C6 74C7                294             jz      normalize_fraction      ; Jump if fraction is an unnormal
                         295
00C8 DDD8                296             fstp    st(0)                   ; Remove unnecessary 1.0 from st(0)
                         297     ;
                         298     ;         Calculate the decimal magnitude associated with this number to
                         299     ;       within one order.  This error will always be inevitable due to
                         300     ;       rounding and lost precision.  As a result, we will deliberately fail
                         301     ;       to consider the LOG10 of the fraction value in calculating the order.
                         302     ;       Since the fraction will always be 1 <= F < 2, its LOG10 will not change
                         303     ;       the basic accuracy of the function.  To get the decimal order of magnitude,
                         304     ;       simply multiply the power of two by LOG10(2) and truncate the result to
                         305     ;       an integer.
                         306     ;
00CA                     307     normal_value:
00CA                     308     not_psuedo_zero:
                         309
00CA DB7EF0              310             fstp    fraction                ; Save the fraction field for later use
00CD DF56FC              311             fist    power_two               ; Save power of two
00D0 D9EC                312             fldlg2                          ; Get LOG10(2)
                         313                                             ; Power_two is now safe to use
00D2 DEC9                314             fmul                            ; Form LOG10(of exponent of number)
00D4 DF5EFA              315             fistp   power_ten               ; Any rounding mode will work here
                         316     ;
                         317     ;         Check if the magnitude of the number rules out treating it as
                         318     ;       an integer.
                         319     ;
                         320     ;       CX has the maximum number of decimal digits allowed.
                         321     ;
```

Figure 15-6 Floating-Point to ASCII Conversion Routine (continued)

```
LOC  OBJ                  LINE    SOURCE

00D7 9B                    322             fwait                           ; Wait for power_ten to be valid
00D8 8B46FA                323             mov     ax,power_ten            ; Get power of ten of value
00DB 2BC1                  324             sub     ax,cx                   ; Form scaling factor necessary in ax
00DD 7722                  325             ja      adjust_result           ; Jump if number will not fit
                           326     ;
                           327     ;         The number is between 1 and 10**(field_size).
                           328     ;       Test if it is an integer.
                           329     ;
00DF DF46FC                330             fild    power_two               ; Restore original number
00E2 8BF2                  331             mov     si,dx                   ; Save return value
00E4 80EAFE                332             sub     dl,NORMAL-EXACT         ; Convert to exact return value
00E7 DB6EF0                333             fld     fraction
00EA D9FD                  334             fscale                          ; Form full value, this is safe here
00EC DDD1                  335             fst     st(1)                   ; Copy value for compare
00EE D9FC                  336             frndint                         ; Test if its an integer
00F0 D8D9                  337             fcomp                           ; Compare values
00F2 9BDD7EFE              338             fstsw   status                  ; Save status
00F6 F746FE0040            339             test    status,4000H            ; C3=1 implies it was an integer
00FB 7550                  340             jnz     convert_integer
                           341
00FD DDD8                  342             fstp    st(0)                   ; Remove non integer value
00FF 8BD6                  343             mov     dx,si                   ; Restore original return value
                           344     ;
                           345     ;         Scale the number to within the range allowed by the BCD format.
                           346     ;       The scaling operation should produce a number within one decimal order
                           347     ;       of magnitude of the largest decimal number representable within the
                           348     ;       given string width.
                           349     ;
                           350     ;         The scaling power of ten value is in ax.
                           351     ;
0101                       352     adjust_result:
                           353
0101 8907                  354             mov     word ptr [bx],ax        ; Set initial power of ten return value
0103 F7D8                  355             neg     ax                      ; Subtract one for each order of
                           356                                             ; magnitude the value is scaled by
0105 E80000          E     357             call    get_power_10            ; Scaling factor is returned as exponent
                           358                                             ; and fraction
0108 DB6EF0                359             fld     fraction                ; Get fraction
010B DEC9                  360             fmul                            ; Combine fractions
010D 8BF1                  361             mov     si,cx                   ; Form power of ten of the maximum
010F D1E6                  362             shl     si,1                    ; BCD value to fit in the string
0111 D1E6                  363             shl     si,1                    ; Index in si
0113 D1E6                  364             shl     si,1
0115 DF46FC                365             fild    power_two               ; Combine powers of two
0118 DEC2                  366             faddp   st(2),st
011A D9FD                  367             fscale                          ; Form full value, exponent was safe
011C DDD9                  368             fstp    st(1)                   ; Remove exponent
                           369     ;
                           370     ;         Test the adjusted value against a table of exact powers of ten.
                           371     ;       The combined errors of the magnitude estimate and power function can
                           372     ;       result in a value one order of magnitude too small or too large to fit
                           373     ;       correctly in the BCD field.  To handle this problem, pretest the
                           374     ;       adjusted value, if it is too small or large, then adjust it by ten and
                           375     ;       adjust the power of ten value
                           376     ;
011E                       377     test_power:
                           378
011E 2EDC940B00      E     379             fcom    power_table[si]+type power_table; Compare against exact power
                           380                                             ; entry   Use the next entry since cx
                           381                                             ; has been decremented by one
0123 9BDFE0                382             fstsw   ax                      ; No wait is necessary
0126 A90041                383             test    ax,4100H                ; If C3 = C0 = 0 then too big
0129 750C                  384             jnz     test_for_small
                           385
012B 2EDE360000      R     386             fidiv   const10                 ; Else adjust value
0130 80E2FD                387             and     dl,not EXACT            ; Remove exact flag
0133 FF07                  388             inc     word ptr [bx]           ; Adjust power of ten value
0135 EB14                  389             jmp     short in_range          ; Convert the value to a BCD integer
                           390
0137                       391     test_for_small:
                           392
0137 2EDC940000      E     393             fcom    power_table[si]         ; Test relative size
013C 9BDFE0                394             fstsw   ax                      ; No wait is necessary
013F A90001                395             test    ax,100H                 ; If C0 = 0 then st(0) >= lower bound
0142 7407                  396             jz      in_range                ; Convert the value to a BCD integer
                           397
0144 2EDE0E0000      R     398             fimul   const10                 ; Adjust value into range
0149 FF0F                  399             dec     word ptr [bx]           ; Adjust power of ten value
                           400
014B                       401     in_range:
                           402
014B D9FC                  403             frndint                         ; Form integer value
                           404     ;
                           405     ;       Assert: 0 <= TOS <= 999,999,999,999,999,999
                           406     ;       The TOS number will be exactly representable in 18 digit BCD format.
                           407     ;
014D                       408     convert_integer:
                           409
014D DF76F0                410             fbstp   bcd_value               ; Store as BCD format number
                           411     ;
                           412     ;         While the store BCD runs, setup registers for the conversion to
                           413     ;       ASCII.
                           414     ;
0150 BE0800                415             mov     si,BCD_SIZE-2           ; Initial BCD index value
```

Figure 15-6 Floating-Point to ASCII Conversion Routine (continued)

```
LOC  OBJ                  LINE     SOURCE

0153 B9040F               416              mov     cx,0F04h                ; Set shift count and mask
0156 BB0100               417              mov     bx,1                    ; Set initial size of ASCII field for sign
0159 8B7E0C               418              mov     di,string_ptr           ; Get address of start of ASCII string
015C 8CD8                 419              mov     ax,ds                   ; Copy ds to es
015E 8EC0                 420              mov     es,ax
0160 FC                   421              cld                             ; Set autoincrement mode
0161 B02B                 422              mov     al,'+'                  ; Clear sign field
0163 F6C201               423              test    dl,MINUS                ; Look for negative value
0166 7402                 424              jz      positive_result
                          425
0168 B02D                 426              mov     al,'-'
                          427
016A                      428      positive_result:
                          429
016A AA                   430              stosb                           ; Bump string pointer past sign
016B 80E2FE               431              and     dl,not MINUS            ; Turn off sign bit
016E 9B                   432              fwait                           ; Wait for fbstp to finish
                          433      ;
                          434      ;         Register usage:
                          435      ;                                  ah:     BCD byte value in use
                          436      ;                                  al:     ASCII character value
                          437      ;                                  dx:     Return value
                          438      ;                                  ch:     BCD mask = 0fh
                          439      ;                                  cl:     BCD shift count = 4
                          440      ;                                  bx:     ASCII string field width
                          441      ;                                  si:     BCD field index
                          442      ;                                  di:     ASCII string field pointer
                          443      ;                                  ds,es:  ASCII string segment base
                          444      ;
                          445      ;         Remove leading zeroes from the number.
                          446      ;
016F                      447      skip_leading_zeroes:
                          448
016F 8A62F0               449              mov     ah,bcd_byte[si]         ; Get BCD byte
0172 8AC4                 450              mov     al,ah                   ; Copy value
0174 D2E8                 451              shr     al,cl                   ; Get high order digit
0176 22C5                 452              and     al,ch                   ; Set zero flag
0178 7516                 453              jnz     enter_odd               ; Exit loop if leading non zero found
                          454
017A 8AC4                 455              mov     al,ah                   ; Get BCD byte again
017C 22C5                 456              and     al,ch                   ; Get low order digit
017E 7518                 457              jnz     enter_even              ; Exit loop if non zero digit found
                          458
0180 4E                   459              dec     si                      ; Decrement BCD index
0181 79EC                 460              jns     skip_leading_zeroes
                          461      ;
                          462      ;         The significand was all zeroes
                          463      ;
0183 B030                 464              mov     al,'0'                  ; Set initial zero
0185 AA                   465              stosb
0186 43                   466              inc     bx                      ; Bump string length
0187 EB16                 467              jmp     short exit_with_value
                          468      ;
                          469      ;         Now expand the BCD string into digit per byte values 0-9
                          470      ;
0189                      471      digit_loop:
                          472
0189 8A62F0               473              mov     ah,bcd_byte[si]         ; Get BCD byte
018C 8AC4                 474              mov     al,ah
018E D2E8                 475              shr     al,cl                   ; Get high order digit
                          476
0190                      477      enter_odd:
                          478
0190 0430                 479              add     al,'0'                  ; Convert to ASCII
0192 AA                   480              stosb                           ; Put digit into ASCII string area
0193 8AC4                 481              mov     al,ah                   ; Get low order digit
0195 22C5                 482              and     al,ch
0197 43                   483              inc     bx                      ; Bump field size counter
                          484
0198                      485      enter_even:
                          486
0198 0430                 487              add     al,'0'                  ; Convert to ASCII
019A AA                   488              stosb                           ; Put digit into ASCII area
019B 43                   489              inc     bx                      ; Bump field size counter
019C 4E                   490              dec     si                      ; Go to next BCD byte
019D 79EA                 491              jns     digit_loop
                          492      ;
                          493      ;         Conversion complete.  Set the string size and remainder.
                          494      ;
019F                      495      exit_with_value:
                          496
019F 8B7E0A               497              mov     di,size_ptr
01A2 891D                 498              mov     word ptr [di],bx
01A4 8BC2                 499              mov     ax,dx                   ; Set return value
01A6 E980FE               500              jmp     exit_proc
                          501
                          502      floating_to_ascii       endp
----                      503      code                    ends
                          504                              end

ASSEMBLY COMPLETE,   NO WARNINGS,   NO ERRORS
```

Figure 15-6 Floating-Point to ASCII Conversion Routine (continued)

```
SERIES-III iAPX286 MACRO ASSEMBLER X108 ASSEMBLY OF MODULE GET_POWER_10
OBJECT MODULE PLACED IN :F3:POW10.OBJ
ASSEMBLER INVOKED BY:  ASM286.86 :F3:POW10.AP2


LOC  OBJ                  LINE     SOURCE

                             1 +1  $title("Calculate the value of 10**ax")
                             2     ;
                             3     ;         This subroutine will calculate the value of 10**ax.
                             4     ;       For values of 0 <= ax < 19, the result will exact.
                             5     ;       All 80286 registers are transparent and the value is returned on
                             6     ;       the TOS as two numbers, exponent in ST(1) and fraction in ST(0).
                             7     ;       The exponent value can be larger than the largest exponent of an
                             8     ;       extended real format number.  Three stack entries are used.
                             9     ;
                            10                     name    get_power_10
                            11
                            12                     public  get_power_10,power_table
                            13
----                        14     stack           stackseg 8
                            15
----                        16     code            segment er public
                            17     ;
                            18     ;         Use exact values from 1.0 to 1e18.
                            19     ;
                            20                     even                            ; Optimize 16 bit access
0000 0000000000000F0        21     power_table     dq      1.0,1e1,1e2,1e3
     3F
0008 00000000000024
     40
0010 00000000000059
     40
0018 0000000000408F
     40
0020 000000000088C3         22                     dq      1e4,1e5,1e6,1e7
     40
0028 00000000006AF8
     40
0030 0000000080842E
     41
0038 00000000D01263
     41
0040 000000008D797          23                     dq      1e8,1e9,1e10,1e11
     41
0048 0000000065CDCD
     41
0050 000000205FA002
     42
0058 000000E8764837
     42
0060 000000A2941A6D         24                     dq      1e12,1e13,1e14,1e15
     42
0068 000040E59C30A2
     42
0070 0000901EC4BCD6
     42
0078 00003426F56B0C
     43
0080 0080E03779C341         25                     dq      1e16,1e17,1e18
     43
0088 00A0D885573476
     43
0090 00C84E676DC1AB
     43
                            26
0098                        27     get_power_10    proc
                            28
0098 3D1200                 29             cmp     ax,18                   ; Test for 0 <= ax < 19
009B 770F                   30             ja      out_of_range
                            31
009D 53                     32             push    bx                      ; Get working index register
009E 8BD8                   33             mov     bx,ax                   ; Form table index
00A0 C1E303                 34             shl     bx,3
00A3 2EDD870000      R      35             fld     power_table[bx]         ; Get exact value
00A8 5B                     36             pop     bx                      ; Restore register value
00A9 D9F4                   37             fxtract                         ; Separate power and fraction
00AB C3                     38             ret                             ; OK to leave fxtract running
                            39     ;
                            40     ;         Calculate the value using the exponentiate instruction.
                            41     ;       The following relations are used:
                            42     ;               10**x = 2**(log2(10)*x)
                            43     ;               2**(I+F) = 2**I * 2**F
                            44     ;               if st(1) = I and st(0) = 2**F then fscale produces 2**(I+F)
                            45     ;
00AC                        46     out_of_range:
                            47
00AC D9E9                   48             fldl2t                          ; TOS = LOG2(10)
00AE C8040000               49             enter   4,0                     ; Format stack
00B2 8946FE                 50             mov     [bp-2],ax               ; Save power of 10 value
00B5 DE4EFE                 51             fimul   word ptr [bp-2]         ; TOS,X = LOG2(10)*P = LOG2(10**P)
00B8 9BD97EFC               52             fstcw   word ptr [bp-4]         ; Get current control word
00BC 8B46FC                 53             mov     ax,word ptr [bp-4]      ; Get control word, no wait necessary
00BF 25FFF3                 54             and     ax,not 0C00H            ; Mask off current rounding field
00C2 0D0004                 55             or      ax,0400H                ; Set round to negative infinity
00C5 8746FC                 56             xchg    ax,word ptr [bp-4]      ; Put new control word in memory
                            57                                             ; old control word is in ax
00C8 D9E8                   58             fld1                            ; Set TOS = -1.0
```

Figure 15-6 Floating-Point to ASCII Conversion Routine (continued)

```
LOC  OBJ                  LINE     SOURCE

00CA D9E0                   59             fchs
00CC D9C1                   60             fld     st(1)                   ; Copy power value in base two
00CE D96EFC                 61             fldcw   word ptr [bp-4]         ; Set new control word value
00D1 D9FC                   62             frndint                         ; TOS = I  -inf < I <= X,  I is an integer
00D3 8946FC                 63             mov     word ptr [bp-4],ax      ; Restore original rounding control
00D6 D96EFC                 64             fldcw   word ptr [bp-4]
00D9 D9CA                   65             fxch    st(2)                   ; TOS = X,  ST(1) = -1.0,  ST(2) = I
00DB D8E2                   66             fsub    st,st(2)                ; TOS,F = X-I.  0 <= TOS < 1.0
00DD 8B46FE                 67             mov     ax,[bp-2]               ; Restore power of ten
00E0 D9FD                   68             fscale                          ; TOS = F/2.  0 <= TOS < 0.5
00E2 D9F0                   69             f2xm1                           ; TOS = 2**(F/2) - 1.0
00E4 C9                     70             leave                           ; Restore stack
00E5 DEE1                   71             fsubr                           ; Form 2**(F/2)
00E7 DCC8                   72             fmul    st,st(0)                ; Form 2**F
00E9 C3                     73             ret                             ; OK to leave fmul running
                            74
                            75     get_power_10    endp
                            76
----                        77     code            ends
                            78                     end

ASSEMBLY COMPLETE,  NO WARNINGS,  NO ERRORS


iAPX286 MACRO ASSEMBLER    Determine TOS register contents                  12:12:13  09/25/83  PAGE    1

SERIES-III iAPX286 MACRO ASSEMBLER X108 ASSEMBLY OF MODULE TOS_STATUS
OBJECT MODULE PLACED IN :F3:TOSST.OBJ
ASSEMBLER INVOKED BY:  ASM286.86 :F3:TOSST.AP2

LOC  OBJ                  LINE     SOURCE

                          1 +1     $title("Determine TOS register contents")
                            2      ;
                            3      ;          This subroutine will return a value from 0-15 in AX corresponding
                            4      ;       to the contents of 80287 TOS.  All registers are transparent and no
                            5      ;       errors are possible.  The return value corresponds to c3,c2,c1,c0
                            6      ;       of FXAM instruction.
                            7      ;
                            8              name    tos_status
                            9
                           10              public  tos_status
                           11
----                       12     stack           stackseg 6      ; Allocate space on the stack
                           13
----                       14     code            segment er public
                           15
0000                       16     tos_status      proc
                           17
0000 D9E5                  18             fxam                    ; Get register contents status
0002 9BDFE0                19             fstsw   ax              ; Get status
0005 8AC4                  20             mov     al,ah           ; Put bit 10-8 into bits 2-0
0007 250740                21             and     ax,4007h        ; Mask out bits c3,c2,c1,c0
000A C0EC03                22             shr     ah,3            ; Put bit c3 into bit 11
000D 0AC4                  23             or      al,ah           ; Put c3 into bit 3
000F B400                  24             mov     ah,0            ; Clear return value
0011 C3                    25             ret
                           26
                           27     tos_status      endp
                           28
----                       29     code            ends
                           30                     end

ASSEMBLY COMPLETE,  NO WARNINGS,  NO ERRORS
```

Function Partitioning

The conversion is imposed by three individual modules. The majority of the conversion's work occurs in the module FLOATING_TO_ASCII. Since the other modules have a more basic use, they are supplied individually. The ASCII to floating-point conversion routine uses the module GET_POWER_10. TOS_STATUS, the other module, indicates what is in the numeric register stack's top, if anything.

Exception Considerations

Producing exceptions is avoided with the utmost care within the function. All numeric values are permitted. Only inadequate space on the numeric register stack would invoke exceptions.

The existence, type (NaN or infinity), and status (unnormal, denormal, zero, sign) of the value passed in the numeric stack is tested, and the string size is checked for minimum and maximum values. The function returns an error code if the register stack's top is empty, or the string size is not large enough.

Within the function, overflow and underflow is evaded for numbers that are too big or too small.

Special Instructions

The operation of multiple numeric instructions, separate data types, and precision control are illustrated by the functions. Instructions for the following are indicated: automatic conversion to BCD, computing the result of 10 raised to an integer value, determining and maintaining concurrency, data synchronization, and utilizing directed rounding on the AMD 80C287 math coprocessor.

This function's double precision accuracy would not be possible with the size and speed of the indicated example if the extended precision data type and built-in exponential function did not exist.

The function is dependent on the numeric BCD data type for conversion from binary floating-point to decimal. BCD digits can be easily unpacked into individual ASCII decimal digits. Most of the work entails reducing the floating-point value to the relatively small range of BCD values. A correct scaling of the given value to an integer between 10^8 and 10^9 is necessary to print a 9-digit result. For example, a scaling factor of 10^9 is necessary for the number +0.123456789 to generate the value +123456789.0, which can be saved in 9 BCD digits. To avoid altering any of the printed digit values, the scale factor must be a power of 10.

These routines should precisely transform all values that can be precisely represented in decimal in the concerned field size. Integer values fitting within the particular string size will be automatically saved into the BCD form, and not scaled. Noninteger values that can be precisely represented in decimal within the bounds of the string size will be precisely transformed, as well. For example, 0.125 can be precisely represented in binary or decimal. The scaling factor necessary to transform this floating-point value to decimal will be 1000, returning 125 as the result. The function, whenever scaling a value, must know where the decimal point is in the last decimal value.

Description of Operation

The conversion process of a floating-point number to decimal ASCII is completed in three main steps. First, determine the number's magnitude. Then scale it for the BCD data type, and finally, convert the BCD data type to a decimal ASCII string.

Determining the result's magnitude necessitates locating the value X represented by $I*10^X$, where $1.0 <= I < 10.0$. To scale the number, multiply it by a scaling factor 10^S; the result should be an integer that does not require any more decimal digits than allowed by the ASCII string.

After the scaling process, the numeric rounding modes and BCD conversion format the number so host software can simply convert it to decimal ASCII.

Great care must be taken in doing each step discussed above. For instance, a numeric meaning does not exist for every floating-point value. The conversion routine should recognize values such as infinity, indefinite, or Not a Number (NaN), and distinctly specify them.

There are special types of numeric values, as well. Since denormals, unnormals, and pseudo zero all establish that accuracy was lost in previous computations, they each have a numeric value and should be recognized.

It is essential that the value be scaled to the BCD range, after the existence of the number's numeric value is determined, and the number is normalized setting proper unnormal flags.

Scaling the Value

It is necessary to identify the number's magnitude in order to scale the number. Determining the magnitude to an accuracy of 1 unit, or within a factor of 10 of the particular value, is adequate. Following the scaling process, an inspection is done to establish whether the result lies within the anticipated range. The result can be altered one decimal order of magnitude in either direction if it is outside this range. The test following scaling must be done because there are unavoidable mistakes in the scaling value.

A quick method is utilized since the magnitude estimate need only be approximate. The estimate is determined by multiplying the power of 2, the unbiased floating-point exponent, affiliated with the number by $\log_{10}2$. Rounding the result to an integer generates an adequately valid estimate. A maximum error of 0.32 may exist in the result if the fraction value is disregarded.

To calculate the scaling factor, use the magnitude of the number string's value and size. This computation is the most unreliable operation of the three-step process. This function uses the equation $10^X = 2^{**}(X * \log_2 10)$ and the exponentiate instruction (F2XM1).

Because of limits on the range of values the F2XM1 instruction permits, the power of 2 value is divided into integer and fraction components. The equation $2^{**}(I+F) = 2^{**}I * 2^{**}F$ lets the FSCALE instruction add the $2^{**}F$ value, computed by F2XM1, and the $2^{**}I$ part once more.

INACCURACY IN SCALING

Placing trailing zeros into the fraction value when removing the integer valued bits lends to the unreliability of these operations. For every integer valued bit in the power of 2 value detached from the fraction bits, one bit of precision is sacrificed in the fraction field because the zeros are placed in the least significant bits.

As many as 14 bits may be missing in the fraction since the greatest floating point exponent value permissible is $2^{14}-1$.

AVOIDING UNDERFLOW AND OVERFLOW

To avoid underflow and overflow when determining the scaling values, the fraction and exponent fields of the number are kept separate. An example of scaling 10^{-4932} to 108 demands a scaling factor of 10^{4950}, which the AMD 80C287 math coprocessor is unable to represent.

Since the exponent and fraction are kept separate, combining the exponents and taking the product of the fractions are separate parts of the scaling operation. The exponent arithmetic concerns small integers, which the AMD 80C287 math coprocessor can represent with ease.

FINAL ADJUSTMENTS

The power function (Get_Power_10) may be able to generate a scaling value that creates a scaled result greater than permitted by the ASCII field. Scaling $9.9999999999999999 \times 10^{4900}$ by $1.0000000000000010 \times 10^{-4883}$, for instance, would generate $1.0000000000000009 \times 10^{18}$. Although the scale factor is in the AMD 80C287 math coprocessor's accuracy range, and the result is in the conversion's accuracy range, BCD format is unable to represent it, which explains why the result's magnitude receives a post-scaling test. Depending on whether the outcome is too small or too great, respectively, it is possible to multiply or divide the result by 10.

Output Format

A binary integer known as the power value specifies the decimal point's position to promote optimum flexibility in output formats. The position of the decimal point is presumed to be to the right of the digit furthest to the right when the power value equals zero. The number of trailing zeros that do not appear is specified by power values above zero. The decimal point's position shifts to the left in the string for every unit less than zero.

Saving the result in BCD and identifying the position of the decimal point is the final process of the conversion. After this step, the BCD string is unpacked into ASCII decimal characters, and the ASCII sign is set according to the initial value's sign.

TRIGONOMETRIC CALCULATION EXAMPLES

The AMD 80C287 instruction set does not come with a full set of trigonometric functions that work directly in computations. Instead, the FPTAN and FPREM instructions supply the standard building blocks for instituting trigonometric functions. Figure 15-7 provides an example of how to implement three trigonometric functions (sine, cosine, and tangent) using the AMD 80C287 math coprocessor. Each function recognizes a legitimate angle argument between -2^{62} and $+2^{62}$.

To compute the result, the above trigonometric functions combine the partial tangent instruction with trigonometric identities. Their accuracy is within 16 units of an extended precision value's lowest 4 bits. The functions are coded for quick execution and minimum size; these may be sacrificed to increase accuracy.

FPTAN and FPREM

These trigonometric functions utilize the AMD 80C287 math coprocessor's FPTAN instruction, in which the angle argument must be between 0 and $\pi/4$ radians (0 to 45 degrees). The FPREM instruction brings the argument within this range. FPREM sets the low three quotient bits that indicate in which octant the initial angle appeared.

A FPREM instruction iteration can decrease angles of 10^{18} radians or lower magnitude to $\pi/4$! Although greater values can be decreased, the result is unreliable since an error in that value's least significant bits means variations of at least 45 degrees in the decreased angle.

Cosine Uses Sine Code

The cosine function uses the majority of the sine function code for the purpose of saving code space. The cosine argument becomes a sine argument as a result of the equation sin ($|A| + \pi/2$) = cos(A). To increase the angle by $\pi/2$, add 010_2 to the FPREM quotient bits that specify the argument's octant.

Increasing the cosine argument by $\pi/2$ if it was considerably much different from $\pi/2$ would be extremely incorrect.

A new relation is utilized in the sine and tangent functions, depending on the particular octant in which the argument falls. The different relations available are indicated in the program listings.

FPTAN creates a ratio that is directly assessed for the tangent function. Depending on the particular octant in which the angle fell, the sine function uses a sine or cosine equation. These operations regularly maintain a divide instruction during execution in order to continue concurrency whenever exiting.

If the range of input angles is limited, such as from 0 to 45 degrees, then there is potential for substantial optimization, because entire angle reduction and octant specification is not imperative.

Each of the three functions start by examining their assigned value. Values involving Not a Number (NaN), infinity, or empty registers must be specially handled. For the FPTAN instruction to execute properly, unnormals must be converted to normal values and denormals must be converted to extremely minute unnormals. The angle's sign is stored to control the result's sign.

Great consideration was taken within the functions to continue simultaneous execution of the AMD 80C287 math coprocessor and host. In effect, the concurrent execution conceals the execution time of the program's decision logic.

Figure 15-7 Calculating Trigonometric Functions

```
SERIES-III iAPX286 MACRO ASSEMBLER X108 ASSEMBLY OF MODULE TRIG_FUNCTIONS
OBJECT MODULE PLACED IN :F3:TRIG.OBJ
ASSEMBLER INVOKED BY:   ASM286.86 :F3:TRIG.AP2


LOC  OBJ                  LINE     SOURCE

                             1 +1  $title("80287 Trignometric Functions")
                             2
                             3                     name    trig_functions
                             4                     public  sine,cosine,tangent
                             5
----                         6     stack           stackseg        6               ; Reserve local space
                             7
   #                         8     sw_287          record  res1:1,cond3:1,top:3,cond2:1,cond1:1,cond0:1,
                             9     &                       res2:8
                            10
----                        11     code            segment er public
                            12     ;
                            13     ;          Define local constants.
                            14     ;
                            15                     even
0000 35C26821A2DA0F         16     pi_quarter      dt      3FFEC90FDAA22168C235R   ; PI/4
     C9FE3F
000A 0000C0FF               17     indefinite      dd      0FFC00000R              ; Indefinite special value
                            18 +1  $eject
```

Figure 15-7 Calculating Trigonometric Functions (continued)

```
LOC  OBJ                    LINE     SOURCE

                              19     ;
                              20     ;       This subroutine calculates the sine or cosine of the angle, given in
                              21     ;     radians.  The angle is in ST(0), the returned value will be in ST(0).
                              22     ;     The result is accurate to within 7 units of the least significant three
                              23     ;     bits of the NPX extended real format.  The PLM/86 definition is:
                              24     ;
                              25     ;     sine:   procedure (angle) real external;
                              26     ;             declare angle real;
                              27     ;             end sine;
                              28     ;
                              29     ;     cosine: procedure (angle) real external;
                              30     ;             declare angle real;
                              31     ;             end cosine;
                              32     ;
                              33     ;       Three stack registers are required.  The result of the function is
                              34     ;     defined as follows for the following arguments:
                              35     ;
                              36     ;                       angle                                   result
                              37     ;
                              38     ;             valid or unnormal less than 2**62 in magnitude    correct value
                              39     ;             zero                                              0 or 1
                              40     ;             denormal                                          correct denormal
                              41     ;             valid or unnormal greater than 2**62              indefinite
                              42     ;             infinity                                          indefinite
                              43     ;             NAN                                               NAN
                              44     ;             empty                                             empty
                              45 +1  $eject
                              46     ;
                              47     ;       This function is based on the NPX fptan instruction.  The fptan
                              48     ;     instruction will only work with an angle of from 0 to PI/4.  With this
                              49     ;     instruction, the sine or cosine of angles from 0 to PI/4 can be accurately
                              50     ;     calculated.  The technique used by this routine can calculate a general
                              51     ;     sine or cosine by using one of four possible operations:
                              52     ;
                              53     ;                       Let R = |angle mod PI/4|
                              54     ;                           B = -1 or 1, according to the sign of the angle
                              55     ;
                              56     ;     1) sin(R)       2) cos(R)       3) sin(PI/4-R)  4) cos(PI/4-R)
                              57     ;
                              58     ;       The choice of the relation and the sign of the result follows the
                              59     ;     decision table shown below based on the octant the angle falls in:
                              60     ;
                              61     ;             octant          sine            cosine
                              62     ;
                              63     ;               0              B*1              2
                              64     ;               1              B*4              3
                              65     ;               2              B*2             -1*1
                              66     ;               3              B*3             -1*4
                              67     ;               4             -B*1             -1*2
                              68     ;               5             -B*4             -1*3
                              69     ;               6             -B*2              1
                              70     ;               7             -B*3              4
                              71     ;
                              72 +1  $eject
                              73     ;
                              74     ;       Angle to sine function is a zero or unnormal.
                              75     ;
000E                          76     sine_zero_unnormal:
                              77
000E DDD9                     78             fstp    st(1)                   ; Remove PI/4
0010 7501                     79             jnz     enter_sine_normalize    ; Jump if angle is unnormal
                              80     ;
                              81     ;       Angle is a zero.
                              82     ;
0012 C3                       83             ret
                              84     ;
                              85     ;       Angle is an unnormal.
                              86     ;
0013                          87     enter_sine_normalize:
                              88
0013 E80901                   89             call    normalize_value
0016 EB2F                     90             jmp     short enter_sine
                              91
0018                          92     cosine  proc                            ; Entry point to cosine
                              93
0018 D9E5                     94             fxam                            ; Look at the value
001A 9BDFE0                   95             fstsw   ax                      ; Store status value
001D 2EDB2E0000      R        96             fld     pi_quarter              ; Setup for angle reduce
0022 B101                     97             mov     cl,1                    ; Signal cosine function
0024 9E                       98             sahf                            ; ZF = C3, PF = C2, CF = C0
0025 7263                     99             jc      funny_parameter         ; Jump if parameter is
                             100                                             ; empty, NAN, or infinity
                             101     ;
                             102     ;       Angle is unnormal, normal, zero, denormal.
                             103     ;
0027 D9C9                    104             fxch                            ; st(0) = angle, st(1) = PI/4
0029 7A1C                    105             jpe     enter_sine              ; Jump if normal or denormal
                             106     ;
                             107     ;       Angle is an unnormal or zero.
                             108     ;
002B DDD9                    109             fstp    st(1)                   ; Remove PI/4
002D 75E4                    110             jnz     enter_sine_normalize
                             111     ;
                             112     ;       Angle is a zero.  cos(0) = 1.0
```

Figure 15-7 Calculating Trigonometric Functions (continued)

```
LOC  OBJ                  LINE     SOURCE

                          113      ;
002F DDD8                 114              fstp    st(0)                   ; Remove 0
0031 D9E8                 115              fld1                            ; Return 1
0033 C3                   116              ret
                          117      ;
                          118      ;         All work is done as a sine function.  By adding PI/2 to the angle
                          119      ;       a cosine is converted to a sine.  Of course the angle addition is not
                          120      ;       done to the argument but rather to the program logic control values
                          121      ;
0034                      122      sine:                                   ; Entry point for sine function
                          123
0034 D9E5                 124              fxam                            ; Look at the parameter
0036 9BDFE0               125              fstsw   ax                      ; Look at fxam status
0039 2ED82E0000     R     126              fld     pi_quarter              ; Get PI/4 value
003E 9E                   127              sahf                            ; CF = C0, PF = C2, ZF = C3
003F 7249                 128              jc      funny_parameter         ; Jump if empty, NAN, or infinity
                          129      ;
                          130      ;           Angle is unnormal, normal, zero, or denormal
                          131      ;
0041 D9C9                 132              fxch                            ; ST(1) = PI/4, st(0) angle
0043 B100                 133              mov     cl,0                    ; Signal sine
0045 7BC7                 134              jpo     sine_zero_unnormal      ; Jump if zero or unnormal
                          135      ;
                          136      ;         ST(0) is either a normal or denormal value.  Both will work.
                          137      ;       Use the fprem instruction to accurately reduce the range of the given
                          138      ;       angle to within 0 and PI/4 in magnitude.  If fprem cannot reduce the
                          139      ;       angle in one shot, the angle is too big to be meaningful, > 2**62
                          140      ;       radians.  Any roundoff error in the calculation of the angle given
                          141      ;       could completely change the result of this function.  It is safest to
                          142      ;       call this very rare case an error.
                          143      ;
0047                      144      enter_sine:
0047 D9F8                 145              fprem                           ; Reduce angle
                          146                                              ; Note that fprem will force a
                          147                                              ; denormal to a very small unnormal
                          148                                              ; Fptan of a very small unnormal
                          149                                              ; will be the same very small
                          150                                              ; unnormal, which is correct
0049 93                   151              xchg    ax,bx                   ; Save old status in BX
004A 9BDFE0               152              fstsw   ax                      ; Check if reduction was complete
                          153                                              ; Quotient in C0,C3,C1
004D 93                   154              xchg    ax,bx                   ; Put new status in bx
004E F6C704               155              test    bh,high(mask cond2)     ; sin(2*N*PI+x) = sin(x)
0051 7544                 156              jnz     angle_too_big
                          157      ;
                          158      ;         Set sign flags and test for which eighth of the revolution the
                          159      ;       angle fell into.
                          160      ;
                          161      ;       Assert: -PI/4 <= st(0) <= PI/4
                          162      ;
0053 D9E1                 163              fabs                            ; Force the argument positive
                          164                                              ; cond1 bit in bx holds the sign
0055 0AC9                 165              or      cl,cl                   ; Test for sine or cosine function
0057 740F                 166              jz      sine_select             ; Jump if sine function
                          167      ;
                          168      ;         This is a cosine function.  Ignore the original sign of the angle
                          169      ;       and add a quarter revolution to the octant id from the fprem instruction.
                          170      ;       cos(A) = sin(A+PI/2) and cos(-A) = cos(A)
                          171      ;
0059 80E4FD               172              and     ah,not high(mask cond1) ; Turn off sign of argument
005C 80CF80               173              or      bh,80H                  ; Prepare to add 010 to C0,C3,C1
                          174                                              ; status value in ax
                          175                                              ; Set busy bit so carry out from
005F 80C740               176              add     bh,high(mask cond3)     ; C3 will go into the carry flag
0062 B000                 177              mov     al,0                    ; Extract carry flag
0064 D0D0                 178              rcl     al,1                    ; Put carry flag in low bit
0066 32F8                 179              xor     bh,al                   ; Add carry to C0 not changing
                          180                                              ; C1 flag
                          181      ;
                          182      ;         See if the argument should be reversed, depending on the octant in
                          183      ;       which the argument fell during fprem.
                          184      ;
0068                      185      sine_select:
                          186
0068 F6C702               187              test    bh,high(mask cond1)     ; Reverse angle if C1 = 1
006B 7404                 188              jz      no_sine_reverse
                          189      ;
                          190      ;         Angle was in octants 1,3,5,7.
                          191      ;
006D DEE9                 192              fsub                            ; Invert sense of rotation
006F EB0E                 193              jmp     short do_sine_fptan     ; 0 < arg <= PI/4
                          194      ;
                          195      ;         Angle was in octants 0,2,4,6.
                          196      ;       Test for a zero argument since fptan will not work if st(0) = 0.
                          197      ;
0071                      198      no_sine_reverse:
                          199
0071 D9E4                 200              ftst                            ; Test for zero angle
0073 91                   201              xchg    ax,cx
0074 9BDFE0               202              fstsw   ax                      ; cond3 = 1 if st(0) = 0
0077 91                   203              xchg    ax,cx
0078 DDD9                 204              fstp    st(1)                   ; Remove PI/4
007A F6C540               205              test    ch,high(mask cond3)     ; If C3=1, argument is zero
007D 7514                 206              jnz     sine_argument_zero
```

Figure 15-7 Calculating Trigonometric Functions (continued)

```
LOC  OBJ                  LINE      SOURCE

                           207      ;
                           208      ;       Assert: 0 < st(0) <= PI/4
                           209      ;
007F                       210      do_sine_fptan:
                           211
007F D9F2                  212              fptan                                   ; TAN ST(0) = ST(1)/ST(0) = Y/X
                           213
0081                       214      after_sine_fptan:
                           215
0081 F6C742                216              test    bh,high(mask cond3 + mask cond1); Look at octant angle fell into
0084 7B1A                  217              jpo     X_numerator                     ; Calculate cosine for octants
                           218                                                      ; 1,2,5,6
                           219      ;
                           220      ;         Calculate the sine of the argument.
                           221      ;       sin(A) = tan(A)/sqrt(1+tan(A)**2)       if tan(A) = Y/X then
                           222      ;       sin(A) = Y/sqrt(X*X + Y*Y)
                           223      ;
0086 D9C1                  224              fld     st(1)                           ; Copy Y value
0088 EB1A                  225              jmp     short finish_sine               ; Put Y value in numerator
                           226      ;
                           227      ;         The top of the stack is either NAN, infinity, or empty.
                           228      ;
008A                       229      funny_parameter:
                           230
008A DDD8                  231              fstp    st(0)                           ; Remove PI/4
008C 7404                  232              jz      return_empty                    ; Return empty if no parm
                           233
008E 7B02                  234              jpo     return_NAN                      ; Jump if st(0) is NAN
                           235      ;
                           236      ;         st(0) is infinity.  Return an indefinite value.
                           237      ;
0090 D9F8                  238              fprem                                   ; ST(1) can be anything
                           239
0092                       240      return_NAN:
0092                       241      return_empty:
                           242
0092 C3                    243              ret                                     ; Ok to leave fprem running
                           244      ;
                           245      ;         Simulate fptan with st(0) = 0
                           246      ;
0093                       247      sine_argument_zero:
                           248
0093 D9E8                  249              fld1                                    ; Simulate tan(0)
0095 EBEA                  250              jmp     after_sine_fptan                ; Return the zero value
                           251      ;
                           252      ;         The angle was too large.  Remove the modulus and dividend from the
                           253      ;       stack and return an indefinite result.
                           254      ;
0097                       255      angle_too_big:
                           256
0097 DED9                  257              fcompp                                  ; Pop two values from the stack
0099 2ED9060A00      R     258              fld     indefinite                      ; Return indefinite
009E 9B                    259              fwait                                   ; Wait for load to finish
009F C3                    260              ret
                           261      ;
                           262      ;         Calculate the cosine of the argument
                           263      ;       cos(A) = 1/sqrt(1+tan(A)**2)      if tan(A) = Y/X then
                           264      ;       cos(A) = X/sqrt(X*X + Y*Y)
                           265      ;
00A0                       266      X_numerator:
                           267
00A0 D9C0                  268              fld     st(0)                           ; Copy X value
00A2 D9CA                  269              fxch    st(2)                           ; Put X in numerator
                           270
00A4                       271      finish_sine:
                           272
00A4 DCC8                  273              fmul    st,st(0)                        ; Form X*X + Y*Y
00A6 D9C9                  274              fxch
00A8 DCC8                  275              fmul    st,st(0)
00AA DEC1                  276              fadd                                    ; st(0) = X*X + Y*Y
00AC D9FA                  277              fsqrt                                   ; st(0) = sqrt(X*X + Y*Y)
                           278
                           279
                           280      ;         Form the sign of the result.  The two conditions are the C1 flag from
                           281      ;       FXAM in bh and the C0 flag from fprem in ah
                           282      ;
00AE 80E701                283              and     bh,high(mask cond0)             ; Look at the fprem C0 flag
00B1 80E402                284              and     ah,high(mask cond1)             ; Look at the fxam C1 flag
00B4 0AFC                  285              or      bh,ah                           ; Even number of flags cancel
00B6 7A02                  286              jpe     positive_sine                   ; Two negatives make a positive
                           287
00B8 D9E0                  288              fchs                                    ; Force result negative
                           289
00BA                       290      positive_sine:
                           291
00BA DEF9                  292              fdiv                                    ; Form final result
00BC C3                    293              ret                                     ; Ok to leave fdiv running
                           294
                           295      cosine  endp
                           296 +1   %eject
                           297      ;
                           298      ;         This function will calculate the tangent of an angle.
                           299      ;       The angle, in radians is passed in ST(0), the tangent is returned
                           300      ;       in ST(0).  The tangent is calculated to an accuracy of 4 units in the
```

Figure 15-7 Calculating Trigonometric Functions (continued)

```
LOC  OBJ                  LINE     SOURCE

                           301     ;       least three significant bits of an extended real format number.  The
                           302     ;       PLM/86 calling format is:
                           303     ;
                           304     ;     tangent:  procedure (angle) real external;
                           305     ;               declare angle real;
                           306     ;               end tangent;
                           307     ;
                           308     ;         Two stack registers are used.  The result of the tangent function is
                           309     ;       defined for the following cases:
                           310     ;
                           311     ;                                    angle                              result
                           312     ;
                           313     ;               valid or unnormal < 2**62 in magnitude          correct value
                           314     ;               0                                               0
                           315     ;               denormal                                        correct denormal
                           316     ;               valid or unnormal > 2**62 in magnitude          indefinite
                           317     ;               NAN                                             NAN
                           318     ;               infinity                                        indefinite
                           319     ;               empty                                           empty
                           320     ;
                           321     ;         The tangent instruction uses the fptan instruction.  Four possible
                           322     ;       relations are used:
                           323     ;
                           324     ;         Let R = |angle MOD PI/4|
                           325     ;             S = -1 or 1 depending on the sign of the angle
                           326     ;
                           327     ;       1) tan(R)       2) tan(PI/4-R)  3) 1/tan(R)     4) 1/tan(PI/4-R)
                           328     ;
                           329     ;         The following table is used to decide which relation to use depending
                           330     ;       on in which octant the angle fell.
                           331     ;
                           332     ;       octant          relation
                           333     ;
                           334     ;         0              S*1
                           335     ;         1              S*4
                           336     ;         2             -S*3
                           337     ;         3             -S*2
                           338     ;         4              S*1
                           339     ;         5              S*4
                           340     ;         6             -S*3
                           341     ;         7             -S*2
                           342     ;
00BD                       343     tangent proc
                           344
00BD D9E5                  345             fxam                                    ; Look at the parameter
00BF 9BDFE0                346             fstsw   ax                              ; Get fxam status
00C2 2ED82E0000      R     347             fld     pi_quarter                      ; Get PI/4
00C7 9E                    348             sahf                                    ; CF = C0, PF = C2, ZF = C3
00C8 72C0                  349             jc      funny_parameter
                           350     ;
                           351     ;         Angle is unnormal, normal, zero, or denormal.
                           352     ;
00CA D9C9                  353             fxch                                    ; st(0) = angle, st(1) = PI/4
00CC 7A17                  354             jpe     tan_zero_unnormal
                           355     ;
                           356     ;         Angle is either an normal or denormal.
                           357     ;       Reduce the angle to the range -PI/4 < result < PI/4
                           358     ;       If fprem cannot perform this operation in one try, the magnitude of the
                           359     ;       angle must be > 2**62.  Such an angle is so large that any rounding
                           360     ;       errors could make a very large difference in the reduced angle.
                           361     ;       It is safest to call this very rare case an error.
                           362     ;
00CE                       363     tan_normal:
                           364
00CE D9F8                  365             fprem                                   ; Quotient in C0,C3,C1
                           366                                                     ; Convert denormals into unnormals
00D0 93                    367             xchg    ax,bx
00D1 9BDFE0                368             fstsw   ax                              ; Quotient identifies octant
                           369                                                     ; original angle fell into
00D4 93                    370             xchg    ax,bx
00D5 F6C704                371             test    bh,high(mask cond2)             ; Test for complete reduction
00D8 75BD                  372             jnz     angle_too_big                   ; Exit if angle was too big
                           373     ;
                           374     ;         See if the angle must be reversed.
                           375     ;
                           376     ;       Assert:  -PI/4 < st(0) < PI/4
                           377     ;
00DA D9E1                  378             fabs                                    ; 0 <= st(0) < PI/4
                           379                                                     ; C3 in bx has the sign flag
00DC F6C702                380             test    bh,high(mask cond1)             ; must be reversed
00DF 740E                  381             jz      no_tan_reverse
                           382     ;
                           383     ;         Angle fell in octants 1,3,5,7.  Reverse it, subtract it from PI/4.
                           384     ;
00E1 DEE9                  385             fsub                                    ; Reverse angle
00E3 EB18                  386             jmp     short do_tangent
                           387     ;
                           388     ;         Angle is either zero or an unnormal.
                           389     ;
00E5                       390     tan_zero_unnormal:
                           391
00E5 DDD9                  392             fstp    st(1)                           ; Remove PI/4
00E7 7405                  393             jz      tan_angle_zero
                           394     ;
                           395     ;         Angle is an unnormal.
```

Figure 15-7 Calculating Trigonometric Functions (continued)

```
LOC  OBJ                  LINE     SOURCE

                           396     ;
00E9 E83300                397             call    normalize_value
00EC EBE0                  398             jmp     tan_normal
                           399
00EE                       400     tan_angle_zero:
                           401
00EE C3                    402             ret
                           403     ;
                           404     ;         Angle fell in octants 0,2,4,6.  Test for st(0) = 0, fptan won't work
                           405     ;
00EF                       406     no_tan_reverse:
                           407
00EF D9E4                  408             ftst                                    ; Test for zero angle
00F1 91                    409             xchg    ax,cx
00F2 9BDFE0                410             fstsw   ax                              ; C3 = 1 if st(0) = 0
00F5 91                    411             xchg    ax,cx
00F6 DDD9                  412             fstp    st(1)                           ; Remove PI/4
00F8 F6C540                413             test    ch,high(mask cond3)
00FB 7515                  414             jnz     tan_zero
                           415
00FD                       416     do_tangent:
                           417
00FD D9F2                  418             fptan                                   ; tan ST(0) = ST(1)/ST(0)
                           419
00FF                       420     after_tangent:
                           421     ;
                           422     ;         Decide on the order of the operands and their sign for the divide
                           423     ;       operation while the fptan instruction is working.
                           424     ;
00FF 8AC7                  425             mov     al,bh                           ; Get a copy of fprem C3 flag
0101 254002                426             and     ax,mask cond1 + high(mask cond3); Examine fprem C3 flag and
                           427                                                     ; FXAM C1 flag
0104 F6C742                428             test    bh,high(mask cond1 + mask cond3); Use reverse divide if in
                           429                                                     ; octants 1,2,5,6
0107 7B0D                  430             jpo     reverse_divide                  ; Note! parity works on low
                           431                                                     ;  8 bits only!
                           432     ;
                           433     ;         Angle was in octants 0,3,4,7.
                           434     ;       Test for the sign of the result.  Two negatives cancel.
                           435     ;
0109 0AC4                  436             or      al,ah
010B 7A02                  437             jpe     positive_divide
                           438
010D D9E0                  439             fchs                                    ; Force result negative
                           440
010F                       441     positive_divide:
                           442
010F DEF9                  443             fdiv                                    ; Form result
0111 C3                    444             ret                                     ; Ok to leave fdiv running
                           445
0112                       446     tan_zero:
                           447
0112 D9E8                  448             fld1                                    ; Force 1/0 = tan(PI/2)
0114 EBE9                  449             jmp     after_tangent
                           450     ;
                           451     ;         Angle was in octants 1,2,5,6.
                           452     ;       Set the correct sign of the result.
                           453     ;
0116                       454     reverse_divide:
                           455
0116 0AC4                  456             or      al,ah
0118 7A02                  457             jpe     positive_r_divide
                           458
011A D9E0                  459             fchs                                    ; Force result negative
                           460
011C                       461     positive_r_divide:
                           462
011C DEF1                  463             fdivr                                   ; Form reciprocal of result
011E C3                    464             ret                                     ; Ok to leave fdiv running
                           465
                           466     tangent endp
                           467     ;
                           468     ;         This function will normalize the value in st(0).
                           469     ;       Then PI/4 is placed into st(1).
                           470     ;
011F                       471     normalize_value:
                           472
011F D9E1                  473             fabs                                    ; Force value positive
0121 D9F4                  474             fxtract                                 ; 0 <= st(0) < 1
0123 D9E8                  475             fld1                                    ; Get normalize bit
0125 DCC1                  476             fadd    st(1),st                        ; Normalize fraction
0127 DEE9                  477             fsub                                    ; Restore original value
0129 D9FD                  478             fscale                                  ; Form original normalized value
012B DDD9                  479             fstp    st(1)                           ; Remove scale factor
012D 2EDB2E0000      R     480             fld     pi_quarter                      ; Get PI/4
0132 D9C9                  481             fxch
0134 C3                    482             ret
                           483
----                       484     code    ends
                           485             end

ASSEMBLY COMPLETE,  NO WARNINGS,  NO ERRORS
```

APPENDIX A

80C286 SYSTEM INITIALIZATION

```
Appendix A  80C286 System Initialization
$title('Switch the 80C286 from Real Address Mode to Protected Mode')
                  name    switch 80C286_modes
                 public   idt_desc,gdt_desc
;
;     Switch the 80C286 from real address mode into protected mode.
;   The initial EPROM GDT, IDT, TSS, and LDT (if any) constructed by BLD286
;   will be copied from EPROM into RAM.  The RAM areas are defined by data
;   segments allocated as fixed entries in the GDT. The CPU registers for
;   GDT, IDT, TSS, and LDT will be set to point at the RAM-based
;   segments. The base fields in the RAM-based GDT will also be updated to
;   point at the RAM-based segments.
;
;     This code is used by adding it to the list of object modules given
;   to BLD286.  BLD286 must then be told to place the segment
;   init_code at address FFFE10H. Execution of the mode switch code begins
;   after RESET. This happens because the mode switch code will start at
;   physical address FFFFF0H, which is the power up address. This code then
;   sets up RAM copies of the EPROM-based segments before jumping to the
;   initial task placed at a fixed GDT entry. After the jump, the CPU
;   executes in the state of the first task defined by BLD286.
;
;     This code will not use any of the EPROM-based tables directly.
;   Such use would result in the 80C286 writing into EPROM to set
;   the A bit. Any use of a GDT or TSS will always be in the RAM copy.
;   The limit and size of the EPROM-based GDT and IDT must be stored at
;   the public symbols idt_desc and gdt_desc. The location commands of BLD286
;   provide this function.
;
;     Interrupts are disabled during this mode switching code. Full error
;   checking is made of the EPROM-based GDT, IDT, TSS, and LDT to assure
;   they are valid before copying them to RAM.  If any of the RAM-based
;   alias segments are smaller than the EPROM segments they are to hold,
;   halt or shutdown will occur. In general, any exception or NMI will
;   cause shutdown to occur until the first task is invoked.
;
;     If the RAM segment is larger than the EPROM segment, the RAM segment
;   will be expanded with zeros. If the initial TSS specifies an LDT,
;   the LDT will also be copied into ldt_alias with zero fill if needed.
;   The EPROM-based or RAM-based GDT, IDT, TSS, and LDT segments may be located
;   anywhere in physical memory.
;
;
```

```
;           Define layout of a descriptor.
;
desc            struc
limit           dw      0           ; Offset of last byte in segment
base_low        dw      0           ; Low 16 bits of 24-bit address
base_high       db      0           ; High 8 bits of 24-bit address
access          db      0           ; Access rights byte
res             dw      0           ; Reserved word
desc            ends
;
;         Define the fixed GDT selector values for the descriptors that
;       define the EPROM-based tables.  BLD286 must be instructed to place the
;       appropriate descriptors into the GDT.
;
gdt_alias       equ     1*size desc ; GDT(1) is data segment in RAM for GDT
idt_alias       equ     2*size desc ; GDT(2) is data segment in RAM for IDT
start_TSS_alias equ     3*size desc ; GDT(3) is data segment in RAM for TSS
start_task      equ     4*size desc ; GDT(4) is TSS for starting task
start_LDT_alias equ     5*size desc ; GDT(5) is data segment in RAM for LDT
;
;         Define machine status word bit positions.
;
PE              equ      1          ; Protection enable
MP              equ      2          ; Monitor processor extension
EM              equ      4          ; Emulate processor extension
;
;         Define particular values of descriptor access rights byte.
;
DT_ACCESS       equ     82H         ; Access byte value for an LDT
DS_ACCESS       equ     92H         ; Access byte value for data segment
                                    ;  which is grow up, at level 0, writeable
TSS_ACCESS      equ     81H         ; Access byte value for an idle TSS
DPL             equ     60H         ; Privilege level field of access rights
ACCESSED        equ     1           ; Define accessed bit
TI              equ     4           ; Position of TI bit
TSS_SIZE        equ     44          ; Size of a TSS
LDT_OFFSET      equ     42          ; Position of LDT in TSS
TIRPL_MASK      equ     size desc-1 ; TI and RPL field mask
;
;         Pass control from the power-up address to the mode switch code.
;       The segment containing this code must be at physical address FFFE10H
;       to place the JMP instruction at physical address FFFFF0H.  The base
;       address is chosen according to the size of this segment.
;
init_code       segment er
cs_offset       equ     0FE10H          ; Low 16 bits of starting address
                org     0FFF0H-cs_offset; Start at address FFFFF0H
                jmp     reset_startup   ; Do not change CS!
;
```

```
;         Define the template for a temporary GDT used to locate the initial
;       GDT and stack.  This data will be copied to location 0.
;       This space is also used for a temporary stack and finally serves
;       as the TSS written into when entering the initial TSS.
;
                org     0               ; Place remaining code below power_up

initial_gdt     desc    <>              ; Filler and null IDT descriptor
gdt_desc        desc    <>              ; Descriptor for EPROM GDT
idt_desc        desc    <>              ; Descriptor for EPROM IDT
temp_desc       desc    <>              ; Temporary descriptor
;
;         Define a descriptor that will point the GDT at location 0.
;       This descriptor will also be loaded into SS to define the initial
;       protected mode stack segment.
;
temp_stack      desc    <end_gdt-initial_gdt-1,0,0,DS_ACCESS,0>
;
;         Define the TSS descriptor used to allow the task switch to the
;       first task to overwrite this region of memory.  The TSS will overlay
;       the initial GDT and stack at location 0.
;
save_tss        desc    <end_gdt-initial_gdt-1,0,0,TSS_ACCESS,0>
;
;         Define the initial stack space and filler for the end of the TSS.
;
                dw      8 dup (0)
end_gdt         label   word

start_pointer   label   dword
                dw      0,start_task    ; Pointer to initial task
;
;         Define template for the task definition list.
;
task_entry      struc                   ; Define layout of task description
TSS_sel         dw      ?               ; Selector for TSS
TSS_alias       dw      ?               ; Data segment alias for TSS
LDT_alias       dw      ?               ; Data segment alias for LDT if any
task_entry      ends

task_list       task_entry      <start_task,start_TSS_alias,start_LDT_alias>
                dw  0                   ; Terminate list

reset_startup:
        cli                             ; No interrupts allowed!
        cld                             ; Use autoincrement mode
        xor     di,di                   ; Point ES:DI at physical address 000000H
        mov     ds,di
        mov     es,di
        mov     ss,di                   ; Set stack at end of reserved area
        mov     sp,end_gdt-initial_gdt
;
```

```
;
;          Form an adjustment factor from the real CS base of FF0000H to the
;        segment base address assumed by ASM286.  Any data reference mode
;        into CS must add an indexing term [BP] to compensate for the difference
;        between the offset generated by ASM286 and the offset required from
;        the base of FF0000H.
;
start    proc                              ; The value of IP at run time will not be
                                           ;  the same as the one used by ASM286!
         call     start1                   ; Get true offset of start1
start1:
         pop      bp
         sub      bp, offset start1        ; Subtract ASM286 offset of start1
                                           ;  leaving adjustment factor in BP
         lidt     initial_gdt[bp]          ; Setup null IDT to force shutdown
                                           ;  on any protection error or interrupt
;
;          Copy the EPROM-based temporary GDT into RAM.
;
         lea      si,initial_gdt[bp]       ; Setup pointer to temporary GDT
                                           ;  template in EPROM
         mov      cx,(end_gdt-initial_gdt)/2  ; Set length
 rep     movs     es:word ptr [di],cs:[si]; Put into reserved RAM area
;
;          Look for AMD 80C287 processor extension.  Assume all ones will be read
;        if an AMD 80C287 is not present.
;
         fninit                            ; Initialize AMD 80C287 if present
         mov      bx,EM                    ; Assume no AMD 80C287
         fstsw    ax                       ; Look at status of AMD 80C287
         or       al,al                    ; No errors should be present
         jnz      set_mode                 ; Jump if no AMD 80C287

         fsetpm                            ; Put AMD 80C287 into protected mode
         mov      bx,MP
;
;          Switch to protected mode and setup a stack, GDT, and LDT.
;
set_mode;
         smsw     ax                       ; Get current MSW
         or       ax,PE                    ; Set PE bit
         or       ax,bx                    ; Set NPX status flags
         lmsw     ax                       ; Enter protected mode!
         jmp      $+2                      ; Clear queue of instructions decoded
                                           ;  while in Real Address Mode
                                           ; CPL is now 0, CS still points at
                                           ; FFFE10 in physical memory
         lgdt     temp_stack[bp]           ; Use initial GDT in RAM area
         mov      ax,temp_stack-initial_gdt   ; Setup SS with valid protected mode
         mov      ss,ax                       ; selector to the RAM GDT and stack
         xor      ax,ax                    ; Set the current LDT to null
         lidt     ax                       ; Any references to it will cause
```

```
                                        ;  an exception causing shutdown
        mov     ax,save_tss-initial_gdt ; Set initial TSS into the low RAM
        ltr     ax                      ; The task switch needs a valid TSS
;
;         Copy the EPROM-based GDT into the RAM data segment alias.
;       First the descriptor for the RAM data segment must be copied into
;       the temporary GDT.
;
        mov     ax,gdt_desc[bp].limit   ; Get size of GDT
        cmp     ax,6*size desc-1        ; Be sure the last entry expected by
                                        ;  this code is inside the GDT
        jb      bad_gdt                 ; Jump if GDT is not big enough

        mov     bx,gdt_desc-initial_gdt ; Form selector to EPROM GDT
        mov     si,gdt_alias            ; Get selector of GDT alias
        call    copy_EPROM_dt           ; Copy into EPROM
        mov     si,idt_alias            ; Get selector of IDT alias
        mov     bx,idt_desc-initial_gdt ; Indicate EPROM IDT
        call    copy_EPROM_dt
        mov     ax,gdt_desc-initial_gdt ; Setup addressing into EPROM GDT
        mov     ds,ax
        mov     bx,gdt_alias            ; Get GDT alias data segment selector
        lgdt    [bx]                    ; Set GDT to RAM GDT
                                        ; SS and TR remain in low RAM
;
;         Copy all task's TSS and LDT segments into RAM
;
        lea     bx,task_list[bp]        ; Define list of tasks to setup
copy_task_loop:
        call    copy_tasks              ; Copy them into RAM
        add     bx,size task_entry      ; Go to next entry
        mov     ax,cs:[bx].tss_sel      ; See if there is another entry
        or      ax,ax
        jnz     copy_task_loop
;
;         With TSS, GDT, and LDT set, startup the initial task!
;
        mov     bx,gdt_alias            ; Point DS at GDT
        mov     ds,bx
        mov     bx,idt_alias            ; Get IDT alias data segment selector
        lidt    [bx]                    ; Set IDT for errors and interrupts
        jmp     start_pointer[bp]       ; Start the first task!
                                        ; The low RAM area is overwritten with
                                        ;  the current CPU context
bad_gdt:
        hlt                             ; Halt here if GDT is not big enough
start   endp
;
```

```
;         Copy the TSS and LDT for the task pointed at by CS:BX.
;       If the task has an LDT it will also be copied down.
;       BX and BP are transparent.
;
bad_tss:
        hlt                             ; Halt here if TSS is invalid
copy_tasks      proc
        mov     si,gdt_alias            ; Get addressability to GDT
        mov     ds,si
        mov     si,cs:[bx].tss_alias    ; Get selector for TSS alias
        mov     es,si                   ; Point ES at alias data segment
        lsl     ax,si                   ; Get length of TSS alias
        mov     si,cs:[bx].tss_sel      ; Get TSS selector
        lar     dx,si                   ; Get alias access rights
        jnz     bad_tss                 ; Jump if invalid reference

        mov     dl,dh                   ; Save TSS descriptor access byte
        and     dh,not DPL              ; Ignore privilege
        cmp     dh,TSS_ACCESS           ; See if TSS
        jnz     bad_tss                 ; Jump if not

        lsl     cx,si                   ; Get length of EPROM based TSS
        cmp     cx,TSS_SIZE-1           ; Verify it is of proper size
        jb      bad_tss                 ; Jump if it is not big enough
;
;         Setup for moving the EPROM-based TSS to RAM
;       DS points at GDT
;
        mov     [si].access,DS_ACCESS   ; Make TSS into data segment
        mov     ds,si                   ; Point DS at EPROM TSS
        call    copy_with_fill          ; Copy DS segment to ES with zero fill
                                        ;  CX has copy count, AX-CX fill count
;
;         Set the GDT TSS limit and base address to the RAM values
;
        mov     ax,gdt_alias            ; Restore GDT addressing
        mov     ds,ax
        mov     es,ax
        mov     di,cs:[bx].tss_sel      ; Get TSS selector
        mov     si,cs:[bx].tss_alias    ; Get RAM alias selector
        movsw                           ; Copy limit
        movsw                           ; Copy low 16 bits of adress
        lodsw                           ; Get high 8 bits of address
        mov     ah,dl                   ; Mark as TSS descriptor
        stosw                           ; Fill in high address and access bytes
        movsw                           ; Copy reserved word
;
;         See if a valid LDT is specified for the startup task
;       If so then copy the EPROM version into the RAM alias.
;
        mov     ds,cs:[bx].tss_alias    ; Address TSS to get LDT
        mov     si,ds:word ptr LDT_OFFSET
```

```
        and     si,not TIRPL_MASK       ; Ignore TI and RPL
        jz      no_ldt                  ; Skip this if no LDT used

        push    si                      ; Save LDT selector
        lar     dx,si                   ; Test descriptor
        jnz     bad_ldt                 ; Jump if invalid selector

        mov     dl,dh                   ; Save LDT descriptor access byte
        and     dh,not DPL              ; Ignore privilege
        cmp     dh, DT_ACCESS           ; Be sure it is an LDT descriptor
        jne     bad_ldt                 ; Jump if invalid

        mov     es:[si].access,DS_ACCESS ; Mark LDT as data segment
        mov     ds,si                   ; Point DS at EPROM LDT
        lsl     ax,si                   ; Get LDT limit
        call    test_dt_limit           ; Verify it is valid
        mov     cx,ax                   ; Save for later
;
;         Examine the LDT alias segment and, if good, copy to RAM
;
        mov     si,cs:[bx].ldt_alias        ; Get ldt alias selector
        mov     es,si                   ; Point ES at alias segment
        lsl     ax,si                   ; Get length of alias segment
        call    test_dt_limit           ; Verify it is valid
        call    copy_with_fill          ; Copy LDT into RAM alias segment
;
;         Set the LDT limit and base address to the RAM copy of the LDT.
;
        mov     si,cs:[bx].ldt_alias    ; Restore LDT alias selector
        pop     di                      ; Restore LDT selector
        mov     ax,gdt_alias            ; Restore GDT addressing
        mov     ds,ax
        mov     es,ax
        movsw                           ; Move the RAM LDT limit
        movsw                           ; Move the low 16 bits across
        lodsw                           ; Get the high 8 bits
        mov     ah,dl                   : Mark as LDT descriptor
        stosw                           ; Set high address and access rights
        movsw                           ; Copy reserved word
no_ldt:
        ret                             ; All done
bad_ldt:
        hlt                             ; Halt here if LDT is invalid
copy_tasks      endp
;
;         Test the descriptor table size in AX to verify that it is an
;       even number of descriptors in length.
;
test_dt_limit   proc
        push    ax                      ; Save length
        end     al,7                    ; Look at low order bits
        cmp     al,7                    ; Must be all ones
```

```
        pop     ax                      ; Restore length
        jne     bad_dt_limit
        ret                             ; All OK
bad_dt_limit  :
        hlt                             ; Die!

test_dt_limit   endp
;
;         Copy the EPROM DT at selector BX in the temporary GDT to the alias
;       data segment at selector SI.  Any improper descriptors or limits
;       will cause shutdown!
;
copy_EPROM_dt   proc

        mov     ax,ss                   ; Point ES:DI at temporary descriptor
        mov     es,ax
        mov     es:[bx].access,DS_ACCESS ; Mark descriptor as a data segment
        mov     es:[bx].res,0           ; Clear reserved word
        lsl     ax,bx                   ; Get limit of EPROM DT
        mov     cx,ax                   ; Save for later
        call    test_dt_limit           ; Verify it is a proper limit
        mov     di,gdt_desc-initial_gdt ; Address EPROM GDT in DS
        mov     ds,di
        mov     di,temp_desc-initial_gdt ; Get selector for temporary descriptor
        push    di                      ; Save offset for later use as selector
        lodsw                           ; Get alias segment size
        call    test_dt_limit           ; Verify it is an even multiple of
                                        ;  descriptors in length
        stosw                           ; Put length into temporary
        movsw                           ; Copy remaining entries into temporary
        movsw
        movsw
        pop     es                      ; ES now points at the GDT alias area
        mov     ds,bx                   ; DS now points at EPROM DT as data
                                        ; Copy segment to alias with zero fill
                                        ; CX is copy count, AX-CX is fill count
                                        ; Fall into copy_with_fill

copy_EPROM_dt   endp
;
;         Copy the segment at DS to the segment at ES for length CX.
;       Fill the end with AX-CX zeros.  Use word operations for speed but
;       allow odd byte operations.
;
copy_with_fill  proc

        xor     si,si                   ; Start at beginning of segments
        xor     di,di
        sub     ax,cx                   ; Form fill count
        add     cx,1                    ; Convert limit to count
        rcr     cx,1                    ; Allow full 64K move
```

```
    rep movsw                           ; Copy DT into alias area
        xchg    ax,cx                   ; Get fill count and zero AX
        jnc     even_copy               ; Jump if even byte count on copy

        movsb                           ; Copy odd byte
        or      cx,cx
        jz      exit_copy               ; Exit if no fill

        stosb                           ; Even out the segment offset
        dec     cx                      ; Adjust remaining fill count
even_copy:
        shr     cx,1                    ; Form word count on fill
    rep stosw                           ; Clear unused words at end
        jnc     exit_copy               ; Exit if no odd byte remains

        stosb                           ; Clear last odd byte
exit_copy:
        ret

copy_with_fill  endp
init_code       ends
                end


$B
```

THE 80C286 INSTRUCTION SET

This section presents the 80C286 instruction set notation. All possible operand types are shown. Instructions are organized alphabetically according to generic operations. Within each operation, many different instructions are possible depending on the operand. The pages are presented in a standardized format, the elements of which are described in the following paragraphs.

Opcode

This column gives the complete object code produced for each form of the instruction. Where possible, the codes are given as hexadecimal bytes, presented in the order in which they will appear in memory. Several shorthand conventions are used for the parts of instructions which specify operands. These conventions are as follows:

/n: (*n* is a digit from 0 through 7) A ModRM byte, plus a possible immediate and displacement field follow the opcode. See Figure B-1 for the encoding of the fields. The digit *n* is the value of the REG field of the ModRM byte. To obtain the possible hexadecimal values for */n,* refer to column *n* of Table B-1. Each row gives a possible value for the effective address operand to the instruction. The entry at the end of the row indicates whether the effective address operand is a register or memory; if memory, the entry indicates what kind of indexing and/or displacement is used. Entries with D8 or D16 signify that a one-byte or two-byte displacement quantity immediately follows the ModRM and optional immediate field bytes. The signed displacement is added to the effective address offset.

/r: A ModRM byte that contains both a register operand and an effective address operand, followed by a possible immediate and displacement field. See Figure B-2 for the encoding of the fields. The ModRM byte could be any value appearing in Table B-1. The column determines which register operand was selected; the row determines the form of effective address. If the row entry mentions D8 or D16, then a one-byte or two-byte displacement follows, as described in the previous paragraph.

cb: A one-byte signed displacement in the range of –128 to +127 follows the opcode. The displacement is sign-extended to 16 bits, and added modulo 65536 to the offset of the instruction FOLLOWING this instruction to obtain the new IP value.

cw: A two-byte displacement is added modulo 65536 to the offset of the instruction FOLLOWING this instruction to obtain the new IP value.

cd: A two-word pointer which will be the new CS:IP value. The offset is given first, followed by the selector.

db: An immediate byte operand to the instruction which follows the opcode and ModRM bytes. The opcode determines if it is a signed value.

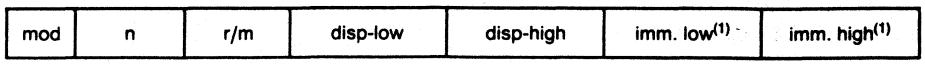
dw: An immediate word operand to the instruction which follows the opcode and ModRM bytes. All words are given in the 80C286 with the low-order byte first.

+rb: A register code from 0 through 7 which is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The codes are: AL = 0, CL = 1, DL = 2, BL = 3, AH = 4, CH = 5, DH = 6, and BH = 7.

+rw: A register code from 0 through 7 which is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The codes are: AX = 0, CX = 1, DX = 2, BX = 3, SP = 4, BP = 5, SI = 6, and DI = 7.

Figure B-1 /n Instruction Byte Format

pp/n Instruction Byte Format



"mod" Field Bit Assignments

mod	Displacement
00	DISP = 0(2), disp-low and disp-high are absent
01	DISP = disp-low sign-extended to 16-bits, disp-high is absent
10	DISP = disp-high: disp-low
11	r/m is treated as a "reg" field

"r/m" Field Bit Assignments

r/m	Operand Address
000	(BX) + (SI) + DISP
001	(BX) + (DI) + DISP
010	(BP) + (SI) + DISP
011	(BP) + (DI) + DISP
100	(SI) + DISP
101	(DI) + DISP
110	(BP) + DISP(2)
111	(BX) + DISP

DISP follows 2nd byte of instruction (before data if required).

NOTES:

1. Opcode indicates presence and size of immediate value.
2. Except if mod = 00 and r/m = 110 then EA = disp-high: disp-low.

Table B-1 ModRM Values

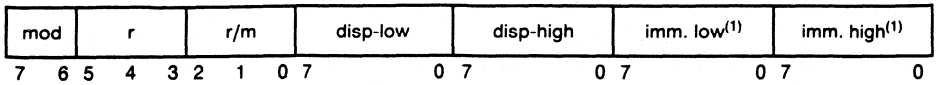
Rb =	AL	CL	DL	BL	AH	CH	DH	BH	
Rw =	AX	CX	DX	BX	SP	BP	SI	DI	
REG =	0	1	2	3	4	5	6	7	
	ModRM values								Effective address
mod=00	00	08	10	18	20	28	30	38	[BX + SI]
	01	09	11	19	21	29	31	39	[BX + DI]
	02	0A	12	1A	22	2A	32	3A	[BP + SI]
	03	0B	13	1B	23	2B	33	3B	[BP + DI]
	04	0C	14	1C	24	2C	34	3C	[SI]
	05	0D	15	1D	25	2D	35	3D	[DI]
	06	0E	16	1E	26	2E	36	3E	D16 (simple var)
	07	0F	17	1F	27	2F	37	3F	[BX]
mod=01	40	48	50	58	60	68	70	78	[BX + SI] + D8(1)
	41	49	51	59	61	69	71	79	[BX + DI] + D8
	42	4A	52	5A	62	6A	72	7A	[BP + SI] + D8
	43	4B	53	5B	63	6B	73	7B	[BP + DI] + D8
	44	4C	54	5C	64	6C	74	7C	[SI] + D8
	45	4D	55	5D	65	6D	75	7D	[DI] + D8
	46	4E	56	5E	66	6E	76	7E	[BP] + D8(2)
	47	4F	57	5F	67	6F	77	7F	[BX] + D8
mod=10	80	88	90	98	A0	A8	B0	B8	[BX + SI] + D16(3)
	81	89	91	99	A1	A9	B1	B9	[BX + DI] + D16
	82	8A	92	9A	A2	AA	B2	BA	[BP +SI] + D16
	83	8B	93	9B	A3	AB	B3	BB	[BP + DI] + D16
	84	8C	94	9C	A4	AC	B4	BC	[SI] + D16
	85	8D	95	9D	A5	AD	B5	BD	[DI] + D16
	86	8E	96	9E	A6	AE	B6	BE	[BP] + D16(2)
	87	8F	97	9F	A7	AF	B7	BF	[BX] + D16
mod=11	C0	C8	D0	D8	E0	E8	F0	F8	Ew=AX Eb=AL
	C1	C9	D1	D9	E1	E9	F1	F9	Ew=CX Eb=CL
	C2	CA	D2	DA	E2	EA	F2	FA	Ew=DX Eb=DL
	C3	CB	D3	DB	E3	EB	F3	FB	Ew=BX Eb=BL
	C4	CC	D4	DC	E4	EC	F4	FC	Ew=SP Eb=AH
	C5	CD	D5	DD	E5	ED	F5	FD	Ew=BP Eb=CH
	C6	CE	D6	DE	E6	EE	F6	FE	Ew=SI Eb=DH
	C7	CF	D7	DF	E7	EF	F7	FF	Ew=DI Eb=BH

NOTES:

1. D8 denotes an 8-bit displacement following the ModRM byte that is sign-extended and added to the index.
2. Default segment register is SS for effective addresses containing a BP index; DS is for other memory effective addresses.
3. D16 denotes the 16-bit displacement following the ModRM byte that is added to the index.

Figure B-2 /r Instruction Byte Format

/r Instruction Byte Format



"mod" Field Bit Assignments

mod	Displacement
00	DISP = 0(2), disp-low and disp-high are absent
01	DISP = disp-low sign-extended to 16-bits, disp-high is absent
10	DISP = disp-high; disp-low
11	r/m is treated as a "reg" field

"r" Field Bit Assignments

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

"r/m" Field Bit Assignments

r/m	Operand Address
000	(BX) + (SI) + DISP
001	(BX) + (DI) + DISP
010	(BP) + (SI) + DISP
011	(BP) + (DI) + DISP
100	(SI) + DISP
101	(DI) + DISP
110	(BP) + DISP(2)
111	(BX) + DISP

DISP follows 2nd byte of instruction (before data if required).

NOTES:

1. Opcode indicates presence and size of immediate field.
2. Except if mod=00 and r/m=110 then EA=disp-high: disp-low.

Instruction

This column gives the instruction mnemonic and possible operands. The type of operand used will determine the opcode and operand encodings. The following entries list the type of operand which can be encoded in the format shown in the instruction column. The convention is to place the destination operand as the left hand operand. Source-only operands follow the destination operand.

In many cases, the same instruction can be encoded several ways. It is recommended that you use the shortest encoding. The short encodings are provided to save memory space.

cb: a destination instruction offset in the range of 128 bytes before the end of this instruction to 127 bytes after the end of this instruction.

cw: a destination offset within the same code segment as this instruction. Some instructions allow a short form of destination offset. See *cb* type for more information.

cd: a destination address, typically in a different code segment from this instruction. Using the *cd:* address form with call instructions saves the code segment selector.

db: a signed value between –128 and +127 inclusive which is an operand of the instruction. For instructions in which the db is to be combined in some way with a word operand, the immediate value is sign-extended to form a word. The upper byte of the word is filled with the topmost bit of the immediate value.

dw: an immediate word value which is an operand of the instruction.

eb: a byte-sized operand. This is either a byte register or a (possibly indexed) byte memory variable. Either operand location may be encoded in the ModRM field. Any memory addressing mode may be used.

ed: a memory-based pointer operand. Any memory addressing mode may be used. Use of a register addressing mode will cause Exception 6.

ew: a word-sized operand. This is either a word register or a (possibly indexed) word memory variable. Either operand location may be encoded in the ModRM field. Any memory addressing mode may be used.

m: a memory location. Operands in registers do not have a memory address. Any memory addressing mode may be used. Use of a register addressing mode will cause Exception 6.

mb: a memory-based byte-sized operand. Any memory addressing mode may be used.

mw: a memory-based word operand. Any memory addressing mode may be used.

rb: one of the byte registers AL, CL, DL, BL, AH, CH, DH, or BH; *rb* has the value 0,1,2,3,4,5,6, and 7, respectively.

rw: one of the word registers AX, CX, DX, BX, SP, BP, SI, or DI; *rw* has the value 0,1,2,3,4,5,6, and 7, respectively.

xb: a simple byte memory variable without a base or index register. MOV instructions between AL and memory have this optimized form if no indexing is required.

xw: a simple word memory variable without a base or index register. MOV instructions between AX and memory have this optimized form if no indexing is required.

Clocks

This column gives the number of clock cycles that this form of the instruction takes to execute. The amount of time for each clock cycle is computed by dividing one microsecond by the number of MHz at which the 80C286 is running. For example, a 10-MHz 80C286 (with the CLK pin connected to a 20-MHz crystal) takes 100 nanoseconds for each clock cycle.

Add one clock to instructions that use the base plus index plus displacement form of addressing. Add two clocks for each 16-bit memory based operand reference located on an odd physical address. Add one clock for each wait state added to each memory read. Wait states inserted in memory writes or instruction fetches do not necessarily increase execution time.

The clock counts establish the maximum execution rate of the 80C286. With no delays in bus cycles, the actual clock count of an 80C286 program will average 5–10% more than the calculated clock count due to instruction sequences that execute faster than they can be fetched from memory.

Some instruction forms give two clock counts, one unlabeled and one labeled. These counts indicate that the instruction has two different clock times for two different circumstances. Following are the circumstances for each possible label:

mem: The instruction has an operand that can either be a register or a memory variable. The unlabeled time is for the register; the *mem* time is for the memory variable. Also, one additional clock cycle is taken for indexed memory variables for which all three possible indices (base register, index register, and displacement) must be added.

noj: The instruction involves a conditional jump or interrupt. The unlabeled time holds when the jump is made; the *noj* time holds when the jump is not made.

pm: If the instruction takes more time to execute when the 80C286 is in Protected Mode. The unlabeled time is for Real Address Mode; the *pm* time is for Protected Mode.

Description

This is a concise description of the operation performed for this form of the instruction. More details are given later in this chapter.

Flags Modified

This is a list of the flags that are set to a meaningful value by the instruction. If a flag is always set to the same value by the instruction, the value is given ("= 0" or "= 1") after the flag name.

Flags Undefined

This is a list of the flags that have an undefined (meaningless) setting after the instruction is executed.

All flags not modified or undefined are unchanged by the instruction.

Operation

This section fully describes the operation performed by the instruction. For some of the more complicated instructions, suggested usage is also indicated.

Protected Mode Exceptions

The possible exceptions involved with this instruction when running under the 80C286 Protected Mode are listed below. These exceptions are abbreviated with a pound sign (#) followed by two capital letters and an optional error code in parenthesis. For example, #GP(0) denotes the general protection exception with an error code of zero. The next section describes all of the 80C286 exceptions and the machine state upon entry to the exception.

If you are an applications programmer, consult the documentation provided with your operating system to determine what actions are taken by the system when exceptions occur.

Real Address Mode Exceptions

Since less error checking is performed by the 80C286 when it is in Real Address Mode, there are fewer exceptions in this mode. One exception that is possible in many instructions is #GP(0). Exception 13 is generated whenever a word operand is accessed from effective address 0FFFFH in a segment. This happens because the second byte of the word is considered located at location 10000H, not at location 0, and thus exceeds the segment's addressability limit.

Protection Exceptions

In parallel with the execution of instructions, the protected-mode 80C286 checks all memory references for validity of addressing and type of access. Violation of the memory protection rules built into the processor will cause a transfer of program control to one of the interrupt procedures described in this section. The interrupts have dedicated positions within the Interrupt Descriptor Table, which is shown in Table B-2. The interrupts are referenced within the instruction set pages by a pound sign (#) followed by a two-letter mnemonic and the optional error code in parenthesis.

Table B-2 **Protection Exceptions of the 80C286**

Abbreviation	Interrupt Number	Description
#UD	6	Undefined Opcode
#NM	7	No Math Unit Available
#DF	8	Double Fault
#MP	9	Math Unit Protection Fault
#TS	10	Invalid Task State Segment
#NP	11	Not Present
#SS	12	Stack Fault
#GP	13	General Protection
#MF	16	Math Fault

Error Codes

Some exceptions cause the 80C286 to pass a 16-bit error code to the interrupt procedure. When this happens, the error code is the last item pushed onto the stack before control is transferred to the interrupt procedure. If stacks were switched as a result of the interrupt (causing a privilege change or task switch), the error code appears on the interrupt procedure's stack, not on the stack of the task that was interrupted.

The error code generally contains the selector of the segment that caused the protection violation. The RPL field (bottom two bits) of the error code does not, however, contain the privilege level. Instead, it contains the following information:

- Bit 0 contains the value 1 if the exception was detected during an interrupt caused by an event external to the program (i.e., an external interrupt, a single step, a processor extension not-present exception, or a processor extension segment overrun). Bit 0 is 0 if the exception was detected while processing the regular instruction stream, even if the instruction stream is part of an external interrupt handling procedure or task. If bit 0 is set, the instruction pointed to by the saved CS:IP address is not responsible for the error. The current task can be restarted unless this is Exception 9.
- Bit 1 is 1 if the selector points to the Interrupt Descriptor Table. In this case, bit 2 can be ignored, and bits 3-10 contain the index into the IDT.
- Bit 1 is 0 if the selector points to the Global or Local Descriptor Tables. In this case, bits 2–15 have their usual selector interpretation: bit 2 selects the table (1 = Local, 0 = Global), and bits 3–15 are the index into the table.

In some cases the 80C286 chooses to pass an error code with no information in it. In these cases, all 16 bits of the error code are zero.

The existence and type of error codes are described under each of the following individual exceptions.

#DF 8 Double Fault (Zero Error Code)

This exception is generated when a second exception is detected while the processor is attempting to transfer control to the handler for an exception. For instance, it is generated if the code segment containing the exception handler is marked not present. It is also generated if invoking the exception handler causes a stack overflow.

This exception is not generated during the execution of an exception handler. Faults detected within the instruction stream are handled by regular exceptions.

The error code is normally zero. The saved CS:IP will point at the instruction that was attempting to execute when the double fault occurred. Since the error code is normally zero, no information on the source of the exception is available. Restart is not possible.

The double-fault exception does not occur when detecting a new exception while trying to invoke handlers for the following exceptions: 1, 2, 3, 4, 5, 6, 7, 9, and 16.

If another exception is detected while attempting to perform the double fault exception, the 80C286 will enter shutdown.

#GP 13 General Protection (Selector or Zero Error Code)

This exception is generated for all protection violations not covered by the other exceptions in this section. Examples of this include:

1. An attempt to address a memory location by using an offset that exceeds the limit for the segment involved.
2. An attempt to jump to a data segment.
3. An attempt to load SS with a selector for a read-only segment.
4. An attempt to write to a read-only segment.
5. Exceeding the maximum instruction length of 10 bytes.

If #GP occurred while loading a descriptor, the error code passed contains the selector involved. Otherwise, the error code is zero.

If the error code is not zero, the instruction can be restarted if the erroneous condition is rectified. If the error code is zero either a limit violation, a write protect violation, or an illegal use of invalid segment register occurred. An invalid segment register contains the values 0–3. A write protect fault on ADC, SBB, RCL, RCR, or XCHG is not restartable.

#MF 16 Math Fault (No Error Code)

This exception is generated when the numeric processor extension (the AMD 80C287 math coprocessor) detects an error signaled by the ERROR input pin leading from the AMD 80C287 math coprocessor to the 80C286. The ERROR pin is tested at the beginning of most floating point instructions, and when a WAIT instruction is executed with the EM bit of the Machine Status Word set to 0 (i.e., no emulation of the math unit). The floating point instructions that do not cause the ERROR pin to be tested are FNCLEX, FNINIT, FSETPM, FNSTCW, FNSTSW, FNSAVE, and FNSTENV.

If the handler corrects the error condition causing the exception, the floating point instruction that caused #MF can be restarted. This is not accomplished by IRET, however, since the fault occurs at the floating point instruction that follows the offending instruction. Before restarting the numeric instruction, the handler must obtain from the AMD 80C287 math coprocessor the address of the offending instruction and the address of the optional numeric operand.

#MP 9 Math Unit Protection Fault (No Error Code)

This exception is generated if the numeric operand is larger than one word and has the second or subsequent words outside the segment's limit. Not all math addressing errors cause Exception 9. If the effective address of an ESCAPE instruction is not in the segment's limit, or if a write is attempted on a read-only segment, or if a one-word operand violates a segment limit, Exception 13 will occur.

The #MP exception occurs during the execution of the numeric instruction by the AMD 80C287 math coprocessor. Thus, the 80C286 may be in an unrelated instruction stream at the time. Exception 9 may occur in a task unrelated to the task that executed the ESC instruction. The operating system should keep track of which task last used the AMD 80C287 math coprocessor.

The offending floating point instruction cannot be restarted; the task which attempted to execute the offending numeric instruction must be aborted. However, if Exception 9 interrupted another task, the interrupted task may be restarted.

The Exception 9 handler must execute FNINIT before executing any ESCAPE or WAIT instruction.

#NM 7 No Math Unit Available (No Error Code)

This exception occurs when any floating point instruction is executed while the EM bit or the TS bit of the Machine Status Word is 1. It also occurs when a WAIT instruction is encountered and both the MP and TS bits of the Machine Status Word are 1.

Depending on the setting of the MSW bits that caused this exception, the exception handler could provide emulation of the AMD 80C287 math coprocessor, or it could perform a context switch of the math processor to prepare it for use by another task.

The instruction causing #NM can be restarted if the handler performs a numeric context switch. If the handler provided emulation of the math unit, it should advance the return pointer beyond the floating point instruction that caused NM.

#NP 11 Not Present (Selector Error Code)

This exception occurs when CS, DS, ES, or the Task Register is loaded with a descriptor that is marked not present but is otherwise valid. It can occur in an LLDT instruction, but the #NP exception will not occur if the processor attempts to load the LDT register during a task switch. A Not Present LDT encountered during a task switch causes the #TS exception.

The error code passed is the selector of the descriptor that is marked not present.

Typically, the Not Present exception handler is used to implement a virtual memory system. The operating system can swap inactive memory segments to a mass-stoage device such as a disk. Applications programs need not be told about this; the next time they attempt to access the swapped-out memory segment, the Not Present handler will be invoked, the segment will be brought back into memory, and the offending instruction within the applications program will be restarted.

If #NP is detected on loading CS, DS, or ES in a task switch, the exception occurs in the new task, and the IRET from the exception handler jumps directly to the next instruction in the new task.

The Not Present exception handler must contain special code to complete the loading of segment registers when #NP is detected in loading the CS or DS registers in a task switch and a trap or interrupt gate was used. The DS and ES registers have been loaded but their descriptors have not been loaded. Any memory reference using the segment register may cause Exception 13. The #NP exception handler should execute code such as the following to ensure full loading of the segment registers:

```
MOV AX,DS
MOV DS,AX
MOV AX,ES
MOV ES,AX
```

#SS 12 Stack Fault (Selector or Zero Error Code)

This exception is generated when a limit violation is detected in addressing through the SS register. It can occur on stack-oriented instructions such as PUSH or POP, as well as other types of memory references using SS such as MOV AX,[BP+28]. It also

can occur on an ENTER instruction when there is not enough space on the stack for the indicated local variable space, even if the stack exception is not triggered by pushing BP or copying the display stack. A stack exception can therefore indicate a stack overflow, a stack underflow or a wild offset. The error code will be zero.

#SS is also generated on an attempt to load SS with a descriptor that is marked not present but is otherwise valid. This can occur in a task switch, an inter-level call, an inter-level return, a move to the SS instruction or a pop to the SS instruction. The error code will be non-zero.

#SS is never generated when addressing through the DS or ES registers even if the offending register points to the same segment as the SS register.

The #SS exception handler must contain special code to complete the loading of segment registers. The DS and ES registers will not be fully loaded if a not-present condition is detected while loading the SS register. Therefore, the #SS exception handler should execute code such as the following to insure full loading of the segment registers:

```
MOV AX,DS
MOV DS,AX
MOV AX,ES
MOV ES,AX
```

Generally, the instruction causing #SS can be restarted, but there is one special case when it cannot: when a PUSHA or POPA instruction attempts to wrap around the 64K boundary of a stack segment. This condition is identified by the value of the saved SP, which can be either 0000H, 0001H, 0FFFEH, or 0FFFFH.

#TS 10 Invalid Task State Segment (Selector Error Code)

This exception is generated during a task switch when the new task state segment is invalid, that is, when a task state segment is too small; when the LDT indicated in a TSS is invalid or not present; when the SS, CS, DS, or ES indicated in a TSS are invalid (task switch); when the back link in a TSS is invalid (inter-task IRET).

#TS is not generated when the SS, CS, DS, or ES back link or privileged stack selectors point to a descriptor that is not present but otherwise is valid. #NP is generated in these cases.

The error code passed to the exception handler contains the selector of the offending segment, which can either be the Task State Segment itself, or a selector found within the Task State Segment.

The instruction causing #TS can be restarted.

#TS must be handled through a task gate.

The exception handler must reset the busy bit in the new TSS.

#UD 6 Undefined Opcode (No Error Code)

This exception is generated when an invalid operation code is detected in the instruction stream. Following are the cases in which #UD can occur:

1. The first byte of an instruction is completely invalid (e.g., 64H).

2. The first byte indicates a 2-byte opcode and the second byte is invalid (e.g., 0FH followed by 0FFH).
3. An invalid register is used with an otherwise valid opcode (e.g., MOV CS,AX).
4. An invalid opcode extension is given in the REG field of the ModRM byte (e.g., 0F6H/1).
5. A register operand is given in an instruction that requires a memory operand (e.g., LGDT AX).

Since the offending opcode will always be invalid, it cannot be restarted. However, the #UD handler might be coded to implement an extension of the 80C286 instruction set. In that case, the handler could advance the return pointer beyond the extended instruction and return control to the program after the extended instruction is emulated. *Any such extensions may be incompatible with an 80386.*

Privilege Level and Task Switching on the 80C286

The 80C286 supports many of the functions necessary to implement a protected, multi-tasking operating system in hardware. This support is provided not by additional instructions, but by extension of the semantics of 8086/8088 instructions that change the value of CS:IP.

Whenever the 80C286 performs an inter-segment jump, call, interrupt, or return, it consults the Access Rights (AR) byte found in the descriptor table entry of the selector associated with the new CS value. The AR byte determines whether the long jump being made is through a gate, or is a task switch, or is a simple long jump to the same privilege level. Table B-3 lists the possible values of the AR byte. The privilege headings at the top of the table give the Descriptor Privilege Level, which is referred to as the DPL within the instruction descriptions.

Each of the CALL, INT, IRET, JMP, and RET instructions contains on its instruction set pages a listing of the access rights checking and actions taken to implement the instruction. Instructions involving task switches contain the symbol SWITCH_TASKS, which is an abbreviation for the following list of checks and actions:

```
SWITCH_TASKS:
 Locked set AR byte of new TSS descriptor to Busy TSS (Bit 1 = 1)
 Current TSS cache must be valid with limit ≥ 41 else #TS (error code will
  be new TSS, but back link points at old TSS)
 Save machine state in current TSS
 If nesting tasks, set the new TSS link to the current TSS selector
 Any exception will be in new context else set the AR byte of current TSS
  descriptor to Available TSS (Bit 1 = 0)
 Set the current TR to selector, base and limit of new TSS
 New TSS limit ≥ 43 else #TS (new TSS)
 Set all machine registers to values from new TSS without loading
  descriptors for DS, ES, CS, SS, LDT
 Clear valid flags for LDT, SS, CS, DS, ES (not valid yet)
 If nesting tasks, set the Nested Task flag to 1
 Set the Task Switched flag to 1
 LDT from the new TSS must be within GDT table limits else #TS(LDT)
 AR byte from LDT descriptor must specify LDT segment else #TS(LDT)
 AR byte from LDT descriptor must indicate PRESENT else #TS(LDT)
 Load LDT cache with new LDT descriptor and set valid bit
 Set CPL to the RPL of the CS selector in the new TSS
 If new stack selector is null #TS(SS)
 SS selector must be within its descriptor table limits else #TS(SS)
 SS selector RPL must be equal to CPL else #TS(SS)
```

```
DPL of SS descriptor must equal to CPL else #TS(SS)
SS descriptor AR byte must indicate writable data segment else #TS(SS)
SS descriptor AR byte must indicate PRESENT else #SS(SS)
Load SS cache with new stack segment and set valid bit
New CS selector must not be null else #TS(CS)
CS selector must be within its descriptor table limits else #TS(CS)
CS descriptor AR byte must indicate code segment else #TS(CS)
If non-conforming then DPL must equal CPL else #TS(CS)
If conforming then DPL must be ≤ CPL else #TS(CS)
CS descriptor AR byte must indicate PRESENT else #NP(CS)
Load CS cache with new code segment descriptor and set valid bit
For DS and ES:
If new selector is not null then perform following checks:
 Index must be within its descriptor table limits else
  #TS(segment selector)
 AR byte must indicate data or readable code else
  #TS(segment selector)
 If data or non-conforming code then:
   DPL must be ≥ CPL else #TS(segment selector)
   DPL must be ≥ RPL else #TS(segment selector)
 AR byte must indicate PRESENT else #NP(segment selector)
 Load cache with new segment descriptor and set valid bit
```

Table B-3 Hexadecimal Values for the Access Rights Byte

Not present, privilege =				Present, privilege =				Descriptor Type
0	1	2	3	0	1	2	3	
00	20	40	60	80	A0	C0	E0	Illegal
01	21	41	61	81	A1	C1	E1	Available Task State Segment
02	22	42	62	82	A2	C2	E2	Local Descriptor Table Segment
03	23	43	63	83	A3	C3	E3	Busy Task State Segment
04	24	44	64	84	A4	C4	E4	Call Gate
05	25	45	65	85	A5	C5	E5	Task Gate
06	26	46	66	86	A6	C6	E6	Interrupt Gate
07	27	47	67	87	A7	C7	E7	Trap Gate
08	28	48	68	88	A8	C8	E8	Illegal
09	29	49	69	89	A9	C9	E9	Illegal
0A	2A	4A	6A	8A	AA	CA	EA	Illegal
0B	2B	4B	6B	8B	AB	CB	EB	Illegal
0C	2C	4C	6C	8C	AC	CC	EC	Illegal
0D	2D	4D	6D	8D	AD	CD	ED	Illegal
0E	2E	4E	6E	8E	AE	CE	EE	Illegal
0F	2F	4F	6F	8F	AF	CF	EF	Illegal
10	30	50	70	90	B0	D0	F0	Expand-up, read only, ignored Data Segment
11	31	51	71	91	B1	D1	F1	Expand-up, read only, accessed Data Segment
12	32	52	72	92	B2	D2	F2	Expand-up, writable, ignored Data Segment
13	33	53	73	93	B3	D3	F3	Expand-up, writable, accessed Data Segment
14	34	54	74	94	B4	D4	F4	Expand-down, read only, ignored Data Segment
15	35	55	75	95	B5	D5	F5	Expand-down, read only, accessed Data Segment
16	36	56	76	96	B6	D6	F6	Expand-down, writable, ignored Data Segment
17	37	57	77	97	B7	D7	F7	Expand-down, writable, accessed Data Segment
18	38	58	78	98	B8	D8	F8	Non-conform, no read, ignored Code Segment
19	39	59	79	99	B9	D9	F9	Non-conform, no read, accessed Code Segment
1A	3A	5A	7A	9A	BA	DA	FA	Non-conform, readable, ignored Code Segment
1B	3B	5B	7B	9B	BB	DB	FB	Non-conform, readable, accessed Code Segment
1C	3C	5C	7C	9C	BC	DC	FC	Conforming, no read, ignored Code Segment
1D	3D	5D	7D	9D	BD	DD	FD	Conforming, no read, accessed Code Segment
1E	3E	5E	7E	9E	BE	DE	FE	Conforming, readable, ignored Code Segment
1F	3F	5F	7F	9F	BF	DF	FF	Conforming, readable, accessed Code Segment

AAA—ASCII Adjust AL After Addition

Opcode	Instruction	Clocks	Description
37	AAA	3	ASCII adjust AL after addition

Flags Modified

Auxiliary carry, carry

Flags Undefined

Overflow, sign, zero, parity

Operation

AAA should be executed only after an ADD instruction which leaves a byte result in the AL register. The lower nibbles of the operands to the ADD instruction should be in the range 0 through 9 (BCD digits). In this case, the AAA instruction will adjust 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 carry flags are set to 0, and AH is unchanged. In any case, AL is left with its top nibble set to 0. To convert AL to an ASCII result, you can follow the AAA instruction with OR AL,30H.

The precise definition of AAA is as follows: if the lower 4 bits of AL are greater than nine, or if the auxiliary carry flag is 1, then increment AL by 6, AH by 1, and set the carry and auxiliary carry flags. Otherwise, reset the carry and auxiliary carry flags. In any case, conclude the AAA operation by setting the upper four bits of AL to zero.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

AAD—ASCII Adjust AX Before Division

Opcode	Instruction	Clocks	Description
D5 0A	AAD	14	ASCII adjust AX before division

FLAGS MODIFIED

Sign, zero, parity

FLAGS UNDEFINED

Overflow, auxiliary carry, carry

OPERATION

AAD is used to prepare two unpacked BCD digits (least significant in AL, most significant in AH) for a division operation which will yield an unpacked result. This is accomplished by setting AL to AL + (10 × AH), and then setting AH to 0. This leaves AX equal to the binary equivalent of the original unpacked 2-digit number.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

AAM—ASCII Adjust AX After Multiply

Opcode	Instruction	Clocks	Description
D4 0A	AAM	15	ASCII adjust AX after multiply

FLAGS MODIFIED

Sign, zero, parity

FLAGS UNDEFINED

Overflow, auxiliary carry, carry

OPERATION

AAM should be used only after executing a MUL instruction between two unpacked BCD digits, leaving the result in the AX register. Since the result is less than one hundred, it is contained entirely in the AL register. AAM unpacks the AL result by dividing AL by ten, leaving the quotient (most significant digit) in AH, and the remainder (least significant digit) in AL.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

AAS—ASCII Adjust AL After Subtraction

Opcode	Instruction	Clocks	Description
3F	AAS	3	ASCII adjust AL after subtraction

FLAGS MODIFIED

Auxiliary carry, carry

FLAGS UNDEFINED

Overflow, sign, zero, parity

OPERATION

AAS should be executed only after a subtraction instruction which left the byte result in the AL register. The lower nibbles of the operands to the SUB instruction should have been in the range 0 through 9 (BCD digits). In this case, the AAS instruction will adjust AL to contain 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 there was no decimal carry, the carry and auxiliary carry flags are set to 0, and AH is unchanged. In any case, AL is left with its top nibble set to 0. To convert AL to an ASCII result, you can follow the AAS instruction with OR AL,30H.

The precise definition of AAS is as follows: if the lower four bits of AL are greater than 9, or if the auxiliary carry flag is 1, then decrement AL by 6, AH by 1, and set the carry and auxiliary carry flags. Otherwise, reset the carry and auxiliary carry flags. In any case, conclude the AAS operation by setting the upper four bits of AL to zero.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

ADC/ADD—Integer Addition

Opcode	Instruction	Clocks	Description
10 /r	ADC *eb,rb*	2,*mem*=7	Add with carry byte register into EA byte
11 /r	ADC *ew,rw*	2,*mem*=7	Add with carry word register into EA word
12 /r	ADC *rb,eb*	2,*mem*=7	Add with carry EA byte into byte register
13 /r	ADC *rw,ew*	2,*mem*=7	Add with carry EA word into word register
14 *db*	ADC AL,*db*	3	Add with carry immediate byte into AL
15 *dw*	ADC AX,*dw*	3	Add with carry immediate word into AX
80 /2 *db*	ADC *eb,db*	3,*mem*=7	Add with carry immediate byte into EA byte
81 /2 *dw*	ADC *ew,dw*	3,*mem*=7	Add with carry immediate word into EA word
83 /2 *db*	ADC *ew,db*	3,*mem*=7	Add with carry immediate byte into EA word
00 /r	ADD *eb,rb*	2,*mem*=7	Add byte register into EA byte
01 /r	ADD *ew,rw*	2,*mem*=7	Add word register into EA word
02 /r	ADD *rb,eb*	2,*mem*=7	Add EA byte into byte register
03 /r	ADD *rw,ew*	2,*mem*=7	Add EA word into word register
04 *db*	ADD AL,*db*	3	Add immediate byte into AL
05 *dw*	ADD AX,*dw*	3	Add immediate word into AX
80 /0 *db*	ADD *eb,db*	3,*mem*=7	Add immediate byte into EA byte
81 /0 *dw*	ADD *ew,dw*	3,*mem*=7	Add immediate word into EA word
83 /0 *db*	ADD *ew,db*	3,*mem*=7	Add immediate byte into EA word

FLAGS MODIFIED

Overflow, sign, zero, auxiliary carry, parity, carry

FLAGS UNDEFINED

None

OPERATION

ADD and ADC perform an integer addition on the two operands. The ADC instruction also adds in the initial state of the carry flag. The result of the addition goes to the first operand. ADC is usually executed as part of a multi-byte or multi-word addition operation.

When a byte immediate value is added to a word operand, the immediate value is first sign-extended.

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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

AND—Logical AND

Opcode	Instruction	Clocks	Description
20 /r	AND *eb,rb*	2,*mem*=7	Logical-AND byte register into EA byte
21 /r	AND *ew,rw*	2,*mem*=7	Logical-AND word register into EA word
22 /r	AND *rb,eb*	2,*mem*=7	Logical-AND EA byte into byte register
23 /r	AND *rw,ew*	2,*mem*=7	Logical-AND EA word into word register
24 *db*	AND AL,*db*	3	Logical-AND immediate byte into AL
25 *dw*	AND AX,*dw*	3	Logical-AND immediate word into AX
80 /4 *db*	AND *eb,db*	3,*mem*=7	Logical-AND immediate byte into EA byte
81 /4 *dw*	AND *ew,dw*	3,*mem*=7	Logical-AND immediate word into EA word

FLAGS MODIFIED

Overflow = 0, sign, zero, parity, carry = 0

FLAGS UNDEFINED

Auxiliary carry

OPERATION

Each bit of the result is a 1 if both corresponding bits of the operands were 1; it is 0 otherwise.

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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

ARPL—Adjust RPL Field of Selector

Opcode	Instruction	Clocks	Description
63 */r*	ARPL *ew,rw*	10,*mem*=11	Adjust RPL of EA word not less than RPL of *rw*

FLAGS MODIFIED

Zero

FLAGS UNDEFINED

None

OPERATION

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 (bottom two bits) of the first operand is less than the RPL field of the second operand, then the zero flag is set to 1 and the RPL field of the first operand is increased to match the second RPL. Otherwise, the zero flag is set to 0 and no change is made to the first operand.

ARPL appears in operating systems software, not in applications programs. It is used to guarantee that a selector parameter to a subroutine does not request more privilege than the caller was entitled to. The second operand used by ARPL would normally be a register that contains the CS selector value of the caller.

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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 6. ARPL is not recognized in Real Address mode.

BOUND—Check Array Index Against Bounds

Opcode	Instruction	Clocks	Description
62 /r	BOUND *rw,md*	noj = 13	INT 5 if *rw* not within bounds

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

BOUND is used to ensure that a signed array index is within the limits defined by a two-word block of memory. The first operand (a register) must be greater than or equal to the first word in memory, and less than or equal to the second word in memory. If the register is not within the bounds, an INTERRUPT 5 occurs.

The two-word block might typically be found just before the array itself and therefore would be accessible at a constant offset of −4 from the array, simplifying the addressing.

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, or ES segments; #SS(0) for an illegal address in the SS segment.

The second operand must be a memory operand, not a register. If the BOUND instruction is executed with a ModRM byte representing a register second operand, then fault #UD will occur.

REAL ADDRESS MODE EXCEPTIONS

INTERRUPT 5 if the bounds test fails, as described above. Interrupt 13 for a second operand at offset 0FFFDH or higher. Interrupt 6 if the second operand is a register, as described in the paragraph above.

CALL—Call Procedure

Opcode	Instruction	Clocks*	Description
E8 *cw*	CALL *cw*	7	Call near, offset relative to next instruction
FF /2	CALL *ew*	7,*mem*=11	Call near, offset absolute at EA word
9A *cd*	CALL *cd*	13,*pm*=26	Call inter-segment, immediate 4-byte address
9A *cd*	CALL *cd*	41	Call gate, same privilege
9A *cd*	CALL *cd*	82	Call gate, more privilege, no parameters
9A *cd*	CALL *cd*	86+4x	Call gate, more privilege, x parameters
9A *cd*	CALL *cd*	177	Call via Task State Segment
9A *cd*	CALL *cd*	182	Call via task gate
FF /3	CALL *ed*	16,*mem*=29	Call inter-segment, address at EA doubleword
FF /3	CALL *ed*	44	Call gate, same privilege
FF /3	CALL *ed*	83	Call gate, more privilege, no parameters
FF /3	CALL *ed*	90+4x	Call gate, more privilege, x parameters
FF /3	CALL *ed*	180	Call via Task State Segment
FF /3	CALL *ed*	185	Call via task gate

*Add one clock for each byte in the next instruction executed.

FLAGS MODIFIED

None, except when a task switch occurs

FLAGS UNDEFINED

None

OPERATION

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 CALL *cw* form of the instruction adds modulo 65536 (the 2-byte operand) to the offset of the instruction following the CALL and sets IP to the resulting offset. The 2-byte offset of the instruction that follows the 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.

The CALL *ew* form of the instruction is the same as CALL *cw* except that the operand specifies a memory location from which the absolute 2-byte offset for the procedure is fetched.

The CALL *cd* form of the instruction uses the 4-byte operand as a pointer to the procedure called. The CALL *ed* form fetches the long pointer from the memory location specified. Both long pointer forms consult the AR byte in the descriptor indexed by the

selector part of the long pointer. The AR byte can indicate one of the following descriptor types:

1. Code Segment—The access rights are checked, the return pointer is pushed onto the stack, and the procedure is jumped to.
2. Call Gate—The offset part of the pointer is ignored. Instead, the entire address of the procedure is taken from the call gate descriptor entry. If the routine being entered is more privileged, then a new stack (both SS and SP) is loaded from the task state segment for the new privilege level, and parameters determined by the word-count field of the call gate are copied from the old stack to the new stack.
3. Task Gate—The current task's context is saved in its Task State Segment (TSS), and the TSS named in the task-gate is used to load the new context. The selector for the outgoing task (from TR) is stored into the new TSS's link field, and the new task's Nested Task flag is set. The outgoing task is left marked busy, the new TSS is marked busy, and execution resumes at the point at which the new task was last suspended.
4. Task State Segment—The current task is suspended and the new task initiated as in 3 above except that there is no intervening gate.

For long calls involving no task switch, the return link is the pointer of the instruction that follows the CALL, that is, the caller's CS and updated IP. Task switches invoked by CALLs are linked by storing the outgoing task's TSS selector in the incoming TSS's link field and setting the Nested Task flag in the new task. Nested tasks must be terminated by an IRET. IRET releases the nested task and follows the back link to the calling task if the NT flag is set.

A precise list of the protection checks made and the actions taken is given by the following list:

```
CALL FAR:
 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:

CALL 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)
 IP must be in code segment limit else #GP(0)
 Load code segment descriptor into CS cache
 Load CS with new code segment selector
 Load IP with new offset

CALL 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)
 IP must be in code segment limit else #GP(0)
 Load code segment descriptor into CS cache
 Load CS with new code segment selector
 Set RPL of CS to CPL
 Load IP with new offset

CALL TO 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

CALL GATE TO 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)
    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:SP value from TSS
    Load new CS:IP value from gate
    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
  Else
    CALL GATE TO SAME PRIVILEGE:
    Stack must have room for 4-byte return address else #SS(0)
    IP must be in code segment limit else #GP(0)
    Load CS:IP from gate
    Push return address onto stack
    Load code segment descriptor into CS-cache
    Set RPL of CS to CPL

CALL TASK GATE:
  Task gate DPL must be ≥ CPL else #GP (gate selector)
  Task gate DPL must be ≥ 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)
    TSS 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 with nesting to TSS
  IP must be in code segment limit else #GP(0)
```

```
TASK STATE SEGMENT:
  TSS DPL must be ≥ CPL else #GP (TSS selector)
  TSS DPL must be ≥ RPL else #GP (TSS selector)
  TSS descriptor AR byte must specify available TSS else #GP (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 #GP(0)

ELSE #GP (code segment selector)
```

PROTECTED MODE EXCEPTIONS

FAR calls: #GP, #NP, #SS, and #TS, as indicated in the list above.

NEAR direct calls: #GP(0) if procedure location is beyond the code segment limits.

NEAR indirect CALL: #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. #GP if the indirect offset obtained is beyond the code segment limits.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

CBW—Convert Byte into Word

Opcode	Instruction	Clocks	Description
98	CBW	2	Convert byte into word (AH = top bit of AL)

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

CBW converts the signed byte in AL to a signed word in AX. It does so by extending the top bit of AL into all of the bits of AH.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

CLC—Clear Carry Flag

Opcode	Instruction	Clocks	Description
F8	CLC	2	Clear carry flag

FLAGS MODIFIED

Carry = 0

FLAGS UNDEFINED

None

OPERATION

CLC sets the carry flag to zero. No other flags or registers are affected.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

CLD—Clear Direction Flag

Opcode	Instruction	Clocks	Description
FC	CLD	2	Clear direction flag, SI and DI will increment

FLAGS MODIFIED

Direction = 0

FLAGS UNDEFINED

None

OPERATION

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.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

CLI—Clear Interrupt Flag

Opcode	Instruction	Clocks	Description
FA	CLI	3	Clear interrupt flag; interrupts disabled

FLAGS MODIFIED

Interrupt = 0

FLAGS UNDEFINED

None

OPERATION

CLI clears the interrupt enable flag if the current privilege level is at least as privileged as IOPL. No other flags are affected. External interrupts will not be recognized at the end of the CLI instruction or thereafter until the interrupt flag is set.

PROTECTED MODE EXCEPTIONS

#GP(0) if the current privilege level is bigger (has less privilege) than the IOPL in the flags register. IOPL specifies the least privileged level at which I/O may be performed.

REAL ADDRESS MODE EXCEPTIONS

None

CLTS—Clear Task Switched Flag

Opcode	Instruction	Clocks	Description
0F 06	CLTS	2	Clear task switched flag

FLAGS MODIFIED

Task switched = 0

FLAGS UNDEFINED

None

OPERATION

CLTS clears the task switched flag in the Machine Status Word. This flag is set by the 80C286 every time a task switch occurs. The TS flag is used to manage processor extensions as follows: every execution of a WAIT or an ESC instruction will be trapped if the MP flag of MSW is set and the task switched flag is set. Thus, if a processor extension is present and a task switch has been made since the last ESC instruction was begun, the processor extension's context must be saved before a new instruction can be issued. The fault routine will save the context and reset the task switched flag or place the task requesting the processor extension into a queue until the current processor extension instruction is completed.

CLTS appears in operating systems software, not in applications programs. It is a privileged instruction that can only be executed at level 0.

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 power-up initialization for Protected Mode)

CMC—Complement Carry Flag

Opcode	Instruction	Clocks	Description
F5	CMC	2	Complement carry flag

FLAGS MODIFIED

Carry

FLAGS UNDEFINED

None

OPERATION

CMC reverses the setting of the carry flag. No other flags are affected.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

CMP—Compare Two Operands

Opcode	Instruction	Clocks	Description
3C *db*	CMP AL,*db*	3	Compare immediate byte from AL
3D *dw*	CMP AX,*dw*	3	Compare immediate word from AX
80 /7 *db*	CMP *eb,db*	3,*mem*=6	Compare immediate byte from EA byte
38 /*r*	CMP *eb,rb*	2,*mem*=7	Compare byte register from EA byte
83 /7 *db*	CMP *ew,db*	3,*mem*=6	Compare immediate byte from EA word
81 /7 *dw*	CMP *ew,dw*	3,*mem*=6	Compare immediate word from EA word
39 /*r*	CMP *ew,rw*	2,*mem*=7	Compare word register from EA word
3A /*r*	CMP *rb,eb*	2,*mem*=6	Compare EA byte from byte register
3B /*r*	CMP *rw,ew*	2,*mem*=6	Compare EA word from word register

FLAGS MODIFIED

Overflow, sign, zero, auxiliary carry, parity, carry

FLAGS UNDEFINED

None

OPERATION

CMP subtracts the second operand from the first operand, but it does not place the result anywhere. Only the flags are changed by this instruction. CMP is usually followed by a conditional jump instruction. See the **Jcond** instructions in this chapter for the list of signed and unsigned flag tests provided by the 80C286.

If a word operand is compared to an immediate byte value, the byte value is first sign-extended.

PROTECTED MODE EXCEPTIONS

#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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

CMPS/CMPSB/CMPSW Compare string operands

Opcode	Instruction	Clocks	Description
A6	CMPSB	8	Compare bytes ES:[DI] from DS:[SI]
A6	CMPS *mb,mb*	8	Compare bytes ES:[DI] from [SI]
A7	CMPSW	8	Compare words ES:[DI] from DS:[SI]

FLAGS MODIFIED

Overflow, sign, zero, auxiliary carry, parity, carry

FLAGS UNDEFINED

None

OPERATION

CMPS compares the byte or word pointed to by SI with the byte or word pointed to by DI by performing the subtraction [SI]−[DI]. The result is not placed anywhere; only the flags reflect the result of the subtraction. The types of the operands to CMPS determine whether bytes or words are compared. The segment addressability of the first (SI) operand determines whether a segment override byte will be produced or whether the default segment register DS is used. The second (DI) operand must be addressible from the ES register; no segment override is possible.

After the comparison is made, both SI and DI 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 was moved; by 2 if a word was moved.

CMPS can be preceded by the REPE or REPNE prefix for block comparison of CX bytes or words. Refer to the REP instruction for details of this operation.

PROTECTED MODE EXCEPTIONS

#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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

CWD—Convert Word to Doubleword

Opcode	Instruction	Clocks	Description
99	CWD	2	Convert word to doubleword (DX:AX = AX)

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

CWD converts the signed word in AX to a signed doubleword in DX:AX. It does so by extending the top bit of AX into all the bits of DX.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

DAA—Decimal Adjust AL After Addition

Opcode	Instruction	Clocks	Description
27	DAA	3	Decimal adjust AL after addition

FLAGS MODIFIED

Sign, zero, auxiliary carry, parity, carry

FLAGS UNDEFINED

Overflow

OPERATION

DAA should be executed only after an ADD instruction which leaves a two-BCD-digit byte result in the AL register. The ADD operands should consist of two packed BCD digits. In this case, the DAA instruction will adjust AL to contain the correct two-digit packed decimal result.

The precise definition of DAA is as follows:

1. If the lower 4 bits of AL are greater than nine, or if the auxiliary carry flag is 1, then increment AL by 6, and set the auxiliary carry flag. Otherwise, reset the auxiliary carry flag.
2. If AL is now greater than 9FH, or if the carry flag is set, then increment AL by 60H, and set the carry flag. Otherwise, clear the carry flag.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

DAS—Decimal Adjust AL After Subtraction

Opcode	Instruction	Clocks	Description
2F	DAS	3	Decimal adjust AL after subtraction

FLAGS MODIFIED

Sign, zero, auxiliary carry, parity, carry

FLAGS UNDEFINED

Overflow

OPERATION

DAS should be executed only after a subtraction instruction which leaves a two-BCD-digit byte result in the AL register. The operands should consist of two packed BCD digits. In this case, the DAS instruction will adjust AL to contain the correct packed two-digit decimal result.

The precise definition of DAS is as follows:

1. If the lower four bits of AL are greater than 9, or if the auxiliary carry flag is 1, then decrement AL by 6, and set the auxiliary carry flag. Otherwise, reset the auxiliary carry flag.
2. If AL is now greater than 9FH, or if the carry flag is set, then decrement AL by 60H, and set the carry flag. Otherwise, clear the carry flag.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

DEC—Decrement by 1

Opcode	Instruction	Clocks	Description
FE /1	DEC *eb*	2,*mem*=7	Decrement EA byte by 1
FF /1	DEC *ew*	2,*mem*=7	Decrement EA word by 1
48+ *rw*	DEC *rw*	2	Decrement word register by 1

FLAGS MODIFIED

Overflow, sign, zero, auxiliary carry, parity

FLAGS UNDEFINED

None

OPERATION

1 is subtracted from the operand. Note that the carry flag is not changed by this instruction. If you want the carry flag set, use the SUB instruction with a second operand of 1.

PROTECTED MODE EXCEPTIONS

#GP(0) if the operand is in a non-writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

DIV—Unsigned Divide

Opcode	Instruction	Clocks	Description
F6 /6	DIV *eb*	14,*mem*= 17	Unsigned divide AX by EA byte
F7 /6	DIV *ew*	22,*mem*= 25	Unsigned divide DX:AX by EA word

FLAGS MODIFIED

None

FLAGS UNDEFINED

Overflow, sign, zero, auxiliary carry, parity, carry

OPERATION

DIV performs an unsigned divide. The dividend is implicit; only the divisor is given as an operand. If the source operand is a BYTE operand, divide AX by the byte. The quotient is stored in AL, and the remainder is stored in AH. If the source operand is a WORD operand, divide DX:AX by the word. The high-order 16 bits of the dividend are kept in DX. The quotient is stored in AX, and the remainder is stored in DX. Non-integral quotients are truncated towards 0. The remainder is always less than the dividend.

PROTECTED MODE EXCEPTIONS

Interrupt 0 if the quotient is too big to fit in the designated register (AL or AX), or if the divisor is zero. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 0 if the quotient is too big to fit in the designated register (AL or AX), or if the divisor is zero. Interrupt 13 for a word operand at offset 0FFFFH.

ENTER Make Stack Frame for Procedure Parameters

Opcode	Instruction	Clocks	Description
C8 *dw* 00	ENTER *dw*,0	11	Make stack frame for procedure parameters
C8 *dw* 01	ENTER *dw*,1	15	Make stack frame for procedure parameters
C8 *dw db*	ENTER *dw*,*db*	12+4*db*	Make stack frame for procedure parameters

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

ENTER is used to create the stack frame required by most block-structured high-level languages. The first operand specifies how many bytes of dynamic storage are to be allocated on the stack for the routine being entered. The second operand gives the lexical nesting level of the routine within the high-level-language source code. It determines how many stack frame pointers are copied into the new stack frame from the preceding frame. BP is used as the current stack frame pointer.

If the second operand is 0, ENTER pushes BP, sets BP to SP, and subtracts the first operand from SP.

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 [BP].

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

```
LEVEL: = LEVEL MOD 32
Push BP
Set a temporary value FRAME_PTR : = SP
If LEVEL > 0 then
 Repeat (LEVEL - 1) times:
  BP : = BP - 2
  Push the word pointed to by BP
 End repeat
 Push FRAME_PTR
End if
BP := FRAME_PTR
SP := SP - first operand
```

PROTECTED MODE EXCEPTIONS

#SS(0) if SP were to go outside of the stack limit within any part of the instruction execution.

REAL ADDRESS MODE EXCEPTIONS

None

HLT—Halt

Opcode	Instruction	Clocks	Description
F4	HLT	2	Halt

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

Successful execution of HLT causes the 80C286 to cease executing instructions and to enter a HALT state. Execution resumes only upon receipt of an enabled interrupt or a reset. If an interrupt is used to resume program execution after HLT, the saved CS:IP value will point to the instruction that follows HLT.

PROTECTED MODE EXCEPTIONS

HLT is a privileged instruction. #GP(0) if the current privilege level is not 0.

REAL ADDRESS MODE EXCEPTIONS

None

IDIV—Signed Divide

Opcode	Instruction	Clocks	Description
F6 /7	IDIV *eb*	17,*mem*=20	Signed divide AX by EA byte (AL = Quo,AH = Rem)
F7 /7	IDIV *ew*	25,*mem*=28	Signed divide DX:AX by EA word (AX = Quo,DX = Rem)

FLAGS MODIFIED

None

FLAGS UNDEFINED

Overflow, sign, zero, auxiliary carry, parity, carry

OPERATION

IDIV performs a signed divide. The dividend is implicit; only the divisor is given as an operand. If the source operand is a BYTE operand, divide AX by the byte. The quotient is stored in AL, and the remainder is stored in AH. If the source operand is a WORD operand, divide DX:AX by the word. The high-order 16 bits of the dividend are in DX. The quotient is stored in AX, and the remainder is stored in DX. Non-integral quotients are truncated towards 0. The remainder has the same sign as the dividend and always has less magnitude than the dividend.

PROTECTED MODE EXCEPTIONS

Interrupt 0 if the quotient is too big 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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 0 if the quotient is too big to fit in the designated register (AL or AX), or if the divisor is 0. Interrupt 13 for a word operand at offset 0FFFFH.

IMUL—Signed Multiply

Opcode	Instruction	Clocks	Description
F6 /5	IMUL *eb*	13,*mem*=16	Signed multiply (AX=AL×EA byte)
F7 /5	IMUL *ew*	21,*mem*=24	Signed multiply (DXAX=AX×EA word)
6B /r *db*	IMUL rw,*db*	21,*mem*=24	Signed multiply imm. byte into word reg.
69 /r *dw*	IMUL *rw,ew,dw*	21,*mem*=24	Signed multiply (*rw*=EA word×imm. word)
6B /r *db*	IMUL *rw,ew,db*	21,*mem*=24	Signed multiply (*rw*=EA word×imm. byte)

FLAGS MODIFIED

Overflow, carry

FLAGS UNDEFINED

Sign, zero, auxiliary carry, parity

OPERATION

IMUL performs signed multiplication. If IMUL has a single byte source operand, then the source is multiplied by AL and the 16-bit signed result is left in AX. Carry and overflow are set to 0 if AH is a sign extension of AL; they are set to 1 otherwise.

If IMUL has a single word source operand, then the source operand is multiplied by AX and the 32-bit signed result is left in DX:AX. DX contains the high-order 16 bits of the product. Carry and overflow are set to 0 if DX is a sign extension of AX; they are set to 1 otherwise.

If IMUL has three operands, then the second operand (an effective address word) is multiplied by the third operand (an immediate word), and the 16 bits of the result are placed in the first operand (a word register). Carry and overflow are set to 0 if the result fits in a signed word (between –32768 and +32767, inclusive); they are set to 1 otherwise.

Note: The low 16 bits of the product of a 16-bit signed multiply are the same as those of an unsigned multiply. The three operand IMUL instruction can be used for unsigned operands as well.

PROTECTED MODE EXCEPTIONS

#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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

IN—Input from Port

Opcode	Instruction	Clocks	Description
E4 *db*	IN AL,*db*	5	Input byte from immediate port into AL
EC	IN AL,DX	5	Input byte from port DX into AL
E5 *db*	IN AX,*db*	5	Input word from immediate port into AX
ED	IN AX,DX	5	Input word from port DX into AX

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

IN transfers a data byte or data word from the port numbered by the second operand into the register (AL or AX) given as the first operand. You can access any port from 0 to 65535 by placing the port number in the DX register then 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 8 bits of the port address will be zero when an 8-bit port I/O is used.

I/O port addresses 00F8H through 00FFH are reserved; they should not be used.

PROTECTED MODE EXCEPTIONS

#GP(0) if the current privilege level is bigger (has less privilege) than IOPL, which is the privilege level found in the flags register.

REAL ADDRESS MODE EXCEPTIONS

None

INC Increment by 1

Opcode	Instruction	Clocks	Description
FE /0	INC *eb*	2,*mem*=7	Increment EA byte by 1
FF /0	INC *ew*	2,*mem*=7	Increment EA word by 1
40+*rw*	INC *rw*	2	Increment word register by 1

FLAGS MODIFIED

Overflow, sign, zero, auxiliary carry, parity

FLAGS UNDEFINED

None

OPERATION

1 is added to the operand. Note that the carry flag is not changed by this instruction. If you want the carry flag set, use the ADD instruction with a second operand of 1.

PROTECTED MODE EXCEPTIONS

#GP(0) if the operand is in a non-writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

INS/INSB/INSW—Input from Port to String

Opcode	Instruction	Clocks	Description
6C	INS *eb*,DX	5	Input byte from port DX into ES:[DI]
6D	INS *ew*,DX	5	Input word from port DX into ES:[DI]
6C	INSB	5	Input byte from port DX into ES:[DI]
6D	INSW	5	Input word from port DX into ES:[DI]

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

INS transfers data from the input port numbered by the DX register to the memory byte or word at ES:DI. The memory operand must be addressable from the ES register; no segment override is possible.

INS does not allow the specification of the port number as an immediate value. The port must be addressed through the DX register.

After the transfer is made, DI is automatically advanced. If the direction flag is 0 (CLD was executed), DI increments; if the direction flag is 1 (STD was executed), DI decrements. DI increments or decrements by 1 if a byte was moved; by 2 if a word was moved.

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.

I/O port addresses 00F8H through 00FFH are reserved; they should not be used.

Note: Not all input port devices can handle the rate at which this instruction transfers input data to memory.

PROTECTED MODE EXCEPTIONS

#GP(0) if CPL > IOPL. #GP(0) if the destination is in a non-writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

INT/INTO—Call to Interrupt Procedure

Opcode	Instruction	Clocks*	Description
CC	INT 3	23**	Interrupt 3 (trap to debugger)
CC	INT 3	40	Interrupt 3, protected mode, same privilege
CC	INT 3	78	Interrupt 3, protected mode, more privilege
CC	INT 3	167	Interrupt 3, protected mode, via task gate
CD *db*	INT *db*	23**	Interrupt numbered by immediate byte
CD *db*	INT *db*	40	Interrupt, protected mode, same privilege
CD *db*	INT *db*	78	Interrupt, protected mode, more privilege
CD *db*	INT *db*	167	Interrupt, protected mode, via task gate
CE	INTO	24,*noj*=3*	Interrupt 4 if overflow flag is 1

* = Add one clock for each byte of the next instruction executed.
** = Real mode

FLAGS MODIFIED

All if a task switch takes place; Trap Flag reset if no task switch takes place. Interrupt Flag is always reset in Real mode, and reset in Protected Mode when INT references an interrupt gate.

FLAGS UNDEFINED

None

OPERATION

The INT instruction generates via software a call to an interrupt procedure. The immediate operand, from 0 to 255, gives the index number into the Interrupt Descriptor Table of the interrupt routine to be called. In protected mode, the IDT consists of 8-byte descriptors; the descriptor for the interrupt invoked must indicate an interrupt gate, a trap gate, or a task gate. In real address mode, the IDT is an array of 4-byte long pointers at the fixed location 00000H.

The INTO instruction is identical to the INT instruction except that the interrupt number is implicitly 4, and the interrupt is made only if the overflow flag of the 80C286 is on. The clock counts for the four forms of INT *db* are valid for INTO, with the number of clocks increased by 1 for the overflow flag test.

The first 32 interrupts are reserved for systems use. Some of these interrupts are exception handlers for internally generated faults. Most of these exception handlers should not be invoked with the INT instruction.

Generally, interrupts behave like far CALLs 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 from the stack.

In Real Address mode, INT pushes the flags, CS and the return IP onto the stack in that order, then resets the Trap Flag, then jumps to the long pointer indexed by the interrupt number, in the interrupt vector table.

In Protected mode, INT also resets the Trap Flag. In Protected mode, the precise semantics of the INT instruction are given by the following:

```
INTERRUPT
 Interrupt vector must be within IDT table limits else #GP (vector
  number x 8 + 2 + EXT)
 Descriptor AR byte must indicate interrupt gate, trap gate, or task gate
   else #GP (vector number x 8 + 2 + EXT)
 If INT instruction then gate descriptor DPL must be ≥ CPL else #GP (vector
  number x 8 + 2 + EXT)
 Gate must be PRESENT else #NP (vector number x 8 + 2 + EXT)
 If 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
  INTERRUPT TO INNER PRIVILEGE:
   Check selector and descriptor for new stack in current Task State
    Segment:
    Selector must be non-null else #TS(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)
   New stack must have room for 10 bytes else #SS(0)
   IP must be in CS limit else #GP(0)
   Load new SS and SP value from TSS
   Load new CS and IP value from gate
   Load CS descriptor
   Load SS descriptor
   Push long pointer to old stack onto new stack
   Push return address onto new stack
   Set CPL to new code segment DPL
   Set RPL of CS to CPL
   If INTERRUPT GATE then set the Interrupts Enabled Flag to 0 (disabled)
   Set the Trap Flag to 0
   Set the Nested Task Flag to 0
 If code segment is conforming or code segment DPL = CPL then
  INTERRUPT TO SAME PRIVILEGE LEVEL:
   Current stack limits must allow pushing 6 bytes else #SS(0)
   If interrupt was caused by fault with error code then
    Stack limits must allow push of two more bytes else #SS(0)
   IP must be in CS limit else #GP(0)
   Push flags onto stack
   Push current CS selector onto stack
   Push return offset onto stack
   Load CS:IP from gate
   Load CS descriptor
   Set the RPL field of CS to CPL
   Push error code (if any) onto stack
   If INTERRUPT GATE then set the Interrupts Enabled Flag to 0 (disabled)
   Set the Trap Flag to 0
   Set the Nested Task Flag to 0

Else #GP (CS selector + EXT)
```

```
If TASK GATE:
 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)
  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 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
 IP must be in CS limit else #GP(0)
```

Note: EXT is 1 if an external event (i.e., a single step, an external interrupt, an MF exception, or an MP exception) caused the interrupt; 0 if not (i.e., an INT instruction or other exceptions).

PROTECTED MODE EXCEPTIONS

#GP, #NP, #SS, and #TS, as indicated in the list above.

REAL ADDRESS MODE EXCEPTIONS

None; the 80C286 will shut down if the SP = 1, 3, or 5 before executing the INT or INTO instruction—due to lack of stack space.

IRET—Interrupt Return

Opcode	Instruction	Clocks	Description
CF	IRET	17,*pm*=31	Interrupt return (far return and pop flags)
CF	IRET	55	Interrupt return, lesser privilege
CF	IRET	169	Interrupt return, different task (NT = 1)

FLAGS MODIFIED

Entire flags register popped from stack

FLAGS UNDEFINED

None

OPERATION

In real address mode, IRET pops IP, CS, and FLAGS from the stack in that order, 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, note that the IOPL bits in the flag register are changed only when CPL = 0.

If NT = 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 of less privilege, IRET then also pops SP and SS from the stack.

If NT = 1, IRET reverses the operation of a CALL or INT that caused a task switch. The task executing IRET has its updated state saved in its Task State Segment. This means that if the task is re-entered, the code that follows IRET will be executed.

The exact checks and actions performed by IRET in protected mode are given on the following page.

```
INTERRUPT RETURN:
 If Nested Task Flag = 1 then
  RETURN FROM NESTED TASK:
    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)
     Task State Segment 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
    IP must be in code segment limit else #GP(0)
 If Nested Task Flag = 0 then
  INTERRUPT RETURN ON STACK:
    Second word on stack must be within stack limits else #SS(0)
    Return CS selector RPL must be ≥ CPL else #GP (Return selector)
    If return selector RPL = CPL then
     INTERRUPT RETURN TO SAME LEVEL:
     Top 6 bytes on stack must be within limits else #SS(0)
     Return CS selector (at SP + 2) 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 = CPL else
  #GP (Return selector)
  If conforming then code segment DPL must be ≤ CPL else
  #GP (Return selector)
  Segment must be PRESENT else #NP (Return selector)
  IP must be in code segment limit else #GP(0)
  Load CS:IP from stack
  Load CS-cache with new code segment descriptor
  Load flags with third word on stack
  Increment SP by 6
Else
INTERRUPT RETURN TO OUTER PRIVILEGE LEVEL:
Top 10 bytes on stack must be within limits else #SS(0)
Examine return CS selector (at SP + 2) 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)
  If conforming then code segment DPL must be > CPL else #GP (Return
  selector)
  Segment must be PRESENT else #NP (Return selector)
Examine return SS selector (at SP + 8) 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 #SS (SS selector)
IP must be in code segment limit else #GP(0)
Load CS:IP from stack
Load flags with values at (SP + 4)
Load SS:SP from stack
Set CPL to the RPL of the return CS selector
Load the CS-cache with the CS descriptor
Load the SS-cache with the SS descriptor
For each of ES and DS:
  If the current register setting is not valid for the outer level,
   then zero the register and clear the valid flag
  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.
```

PROTECTED MODE EXCEPTIONS

#GP, #NP, or #SS, as indicated in the above listing.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 if the stack is popped when it has offset 0FFFFH.

Jcond—Jump Short If Condition Met

Opcode	Instruction	Clocks*	Description
77 *cb*	JA *cb*	7,*noj*=3	Jump short if above (CF = 0 and ZF = 0)
73 *cb*	JAE *cb*	7,*noj* = 3	Jump short if above or equal (CF = 0)
72 *cb*	JB *cb*	7,*noj*=3	Jump short if below (CF = 1)
76 *cb*	JBE *cb*	7,*noj*=3	Jump short if below or equal (CF = 1 or F = 1)
72 *cb*	JC *cb*	7,*noj*=3	Jump short if carry (CF = 1)
E3 *cb*	JCXZ *cb*	8,*noj*=4	Jump short if CX register is zero
74 *cb*	JE *cb*	7,*noj*=3	Jump short if equal (ZF = 1)
7F *cb*	JG *cb*	7,*noj*=3	Jump short if greater (ZF = 0 and SF = OF)
7D *cb*	JGE *cb*	7,*noj*=3	Jump short if greater or equal (SF = OF)
7C *cb*	JL *cb*	7,*noj*=3	Jump short if less (SF/ = OF)
7E *cb*	JLE *cb*	7,*noj*=3	Jump short if less or equal (ZF = 1 or SF/ = OF)
76 *cb*	JNA *cb*	7,*noj*=3	Jump short if not above (CF = 1 or ZF = 1)
72 *cb*	JNAE *cb*	7,*noj*=3	Jump short if not above/equal (CF = 1)
73 *cb*	JNB *cb*	7,*noj*=3	Jump short if not below (CF = 0)
77 *cb*	JNBE *cb*	7,*noj*=3	Jump short if not below/equal (CF = 0 and ZF = 0)
73 *cb*	JNC *cb*	7,*noj*=3	Jump short if not carry (CF = 0)
75 *cb*	JNE *cb*	7,*noj*=3	Jump short if not equal (ZF = 0)
7E *cb*	JNG *cb*	7,*noj*=3	Jump short if not greater (ZF = 1 or SF/ = OF)
7C *cb*	JNGE *cb*	7,*noj*=3	Jump short if not greater/equal (SF = OF)
7D *cb*	JNL *cb*	7,*noj*=3	Jump short if not less (SF = OF)
7F *cb*	JNLE *cb*	7,*noj*=3	Jump short if not less/equal (ZF = 0 and SF = OF)
71 *cb*	JNO *cb*	7,*noj*=3	Jump short if not overflow (OF = 0)
7B *cb*	JNP *cb*	7,*noj*=3	Jump short if not parity (PF = 0)
79 *cb*	JNS *cb*	7,*noj*=3	Jump short if not sign (SF = 0)
75 *cb*	JNZ *cb*	7,*noj*=3	Jump short if not zero (ZF = 0)
70 *cb*	JO *cb*	7,*noj*=3	Jump short if overflow (OF = 1)
7A *cb*	JP *cb*	7,*noj*=3	Jump short if parity (PF = 1)
7A *cb*	JPE *cb*	7,*noj*=3	Jump short if parity even (PF = 1)
7B *cb*	JPO *cb*	7,*noj*=3	Jump short if parity odd (PF = 0)
78 *cb*	JS *cb*	7,*noj*=3	Jump short if sign (SF = 1)
74 *cb*	JZ *cb*	7,*noj*=3	Jump short if zero (ZF = 1)

* When a jump is taken, add one clock for every byte of the next instruction executed.

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

Conditional jumps (except for JCXZ, explained below) test the flags, which presumably have been set in some meaningful way 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 comparing signed integers; above and below are used for unsigned integers.

If the given condition is true, then a short jump is made to the label provided as the operand. Instruction encoding is most efficient when the target for the conditional jump is in the current code segment and within –128 to +127 bytes of the first byte of the next instruction. Alternatively, the opposite sense (e.g., JNZ has opposite sense to that of JZ) of the conditional jump can skip around an unconditional jump to the destination.

This range is necessary for the assembler to construct a one-byte signed displacement from the end of the current instruction. If the label is out-of-range, or if the label is a FAR label, then you must perform a jump with the opposite condition around an unconditional jump to the non-short label.

Because there are, in many instances, several ways to interpret a particular state of the flags, ASM286 provides more than one mnemonic for most of the conditional jump opcodes. For example, consider that a programmer who has just compared a character to another in AL might wish to jump if the two were equal (JE), while another programmer who had just ANDed AX with a bit field mask would prefer to consider only whether the result was zero or not (he would use JZ, a synonym for JE).

JCXZ differs from the other conditional jumps in that it actually tests the contents of the CX register for zero, rather than interrogating the flags. This instruction is useful following a conditionally repeated string operation (e.g., REPE SCASB) or a conditional loop instruction (such as LOOPNE TARGETLABEL). These instructions implicitly use a limiting count in the CX register. Looping (repeating) ends when either the CX register goes to zero or the condition specified in the instruction (flags indicating equals in both of the above cases) occurs. JCXZ is useful when the terminations must be handled differently.

PROTECTED MODE EXCEPTIONS

#GP(0) if the offset jumped to is beyond the limits of the code segment.

REAL ADDRESS MODE EXCEPTIONS

None

JMP—Jump

Opcode	Instruction	Clocks*	Description
EB *cb*	JMP *cb*	7	Jump short
EA *cd*	JMP *cd*	180	Jump to task gate
E9 *cw*	JMP *cw*	7	Jump near
EA *cd*	JMP *cd*	11,*pm*=23	Jump far (4-byte immediate address)
EA *cd*	JMP *cd*	38	Jump to call gate, same privilege
EA *cd*	JMP *cd*	175	Jump via Task State Segment
FF /4	JMP *ew*	7,*mem*=11	Jump near to EA word (absolute offset)
FF /5	JMP *ed*	15,*pm*=26	Jump far (4-byte effective address in memory doubleword)
FF /5	JMP *ed*	41	Jump to call gate, same privilege
FF /5	JMP *ed*	178	Jump via Task State Segment
FF /5	JMP *ed*	183	Jump to task gate

* Add one clock for every byte of the next instruction executed.

FLAGS MODIFIED

All if a task switch takes place; none if no task switch occurs.

FLAGS UNDEFINED

None

OPERATION

The JMP instruction transfers program control to a different instruction stream without recording any return information.

For inter-segment jumps, the destination can be a code segment, a call gate, a task gate, or a Task State Segment. The latter two destinations cause a complete task switch to take place.

Control transfers within a segment use the JMP *cw* or JMP *cb* forms. The operand is a relative offset added modulo 65536 to the offset of the instruction that follows the JMP. The result is the new value of IP; the value of CS is unchanged. The byte operand is sign-extended before it is added; it can therefore be used to address labels within 128 bytes in either direction from the next instruction.

Indirect jumps within a segment use the JMP *ew* form. The contents of the register or memory operand is an absolute offset, which becomes the new value of IP. Again, CS is unchanged.

Inter-segment jumps in real address mode simply set IP to the offset part of the long pointer and set CS to the selector part of the pointer.

In Protected Mode, inter-segment jumps cause the 80C286 to consult the descriptor addressed by the selector part of the long pointer. The AR byte of the descriptor determines the type of the destination. (See Table B-3 for possible values of the AR byte.) Following are the possible destinations:

1. Code segment—The addressability and visibility of the destination are verified, and CS and IP are loaded with the destination pointer values.
2. Call gate—The offset part of the destination pointer is ignored. After checking for validity, the processor jumps to the location stored in the call gate descriptor.
3. Task gate—The current task's state is saved in its Task State Segment (TSS), and the TSS named in the task gate is used to load a new context. The outgoing task is marked not busy, the new TSS is marked busy, and execution resumes at the point at which the new task was last suspended.
4. TSS—The current task is suspended and the new task is initiated as in 3 above except that there is no intervening gate.

Following is the list of checks and actions taken for long jumps in protected mode:

```
JUMP FAR:
 If indirect then check access of EA doubleword #GP(0) or #SS(0) if limit
 violation
 Destination selector is not null else #GP(0)
 Destination selector index is within its descriptor table limits else #GP
 (selector)
 Examine AR byte of destination selector for legal values:

  JUMP CONFORMING CODE SEGMENT:
   Descriptor DPL must be ≤ CPL else #GP (selector)
   Segment must be PRESENT else #NP (selector)
   IP must be in code segment limit else #GP(0)
   Load CS:IP from destination pointer
   Load CS-cache with new segment descriptor

  JUMP NONCONFORMING CODE SEGMENT:
   RPL of destination selector must be ≤ CPL else #GP (selector)
   Descriptor DPL must = CPL else #GP (selector)
   Segment must be PRESENT else #NP (selector)
   IP must be in code segment limit else #GP(0)
   Load CS:IP from destination pointer
   Load CS-cache with new segment descriptor
   Set RPL field of CS register to CPL

  JUMP TO 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, code segment descriptor DPL must = CPL else #GP (CS
    selector)
   If conforming, then code segment descriptor DPL must be ≤ CPL else #GP
    (CS selector)
   Code Segment must be PRESENT else #NP (CS selector)
   IP must be in code segment limit else #GP(0)
   Load CS:IP from call gate
   Load CS-cache with new code segment
   Set RPL of CS to CPL

  JUMP 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
  IP must be in code segment limit else #GP(0)

JUMP 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 with nesting to TS.
  IP must be in code segment limit else #GP(0)

Else GP (selector)
```

PROTECTED MODE EXCEPTIONS

For NEAR jumps, #GP(0) if the destination offset is beyond the limits of the current code segment. For FAR jumps, #GP, #NP, #SS, and #TS, as indicated above. #UD if indirect inter-segment jump operand is a register.

REAL ADDRESS MODE EXCEPTIONS

#UD if indirect inter-segment jump operand is a register.

LAHF—Load Flags into AH Register

Opcode	Instruction	Clocks	Description
9F	LAHF	2	Load: AH = flags SF ZF xx AF xx PF xx CF

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The low byte of the flags word is transferred to AH. The bits, from MSB to LSB, are as follows: sign, zero, indeterminate, auxiliary carry, indeterminate, parity, indeterminate, and carry. See Figure 3-5.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

LAR—Load Access Rights Byte

Opcode	Instruction	Clocks	Description
0F 02 */r*	LAR *rw,ew*	14,*mem*=16	Load: high(*rw*) = Access Rights byte, selector *ew*

FLAGS MODIFIED

Zero

FLAGS UNDEFINED

None

OPERATION

LAR expects the second operand (memory or register word) to contain a selector. If the associated descriptor is visible at the current privilege level and at the selector RPL, then the access rights byte of the descriptor is loaded into the high byte of the first (register) operand, and the low byte is set to zero. The zero flag is set if the loading was performed (i.e., the selector index is within the table limit, descriptor DPL≥CPL, and descriptor DPL≥selector RPL); the zero flag is cleared otherwise.

Selector operands cannot cause protection exceptions.

PROTECTED MODE EXCEPTIONS

#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.

Real Address Mode Exceptions

INTERRUPT 6; LAR is unrecognized in Real Address mode.

LDS/LES—Load Doubleword Pointer

Opcode	Instruction	Clocks	Description
C5 /r	LDS *rw,ed*	7,*pm*=21	Load EA doubleword into DS and word register
C4 /r	LES *rw,ed*	7,*pm*=21	Load EA doubleword into ES and word register

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The four-byte pointer at the memory location indicated by the second operand is loaded into a segment register and a word register. The first word of the pointer (the offset) is loaded into the register indicated by the first operand. The last word of the pointer (the selector) is loaded into the segment register (DS or ES) given by the instruction opcode.

When the segment register is loaded, its associated cache is also loaded. The data for the cache is obtained from the descriptor table entry for the selector given.

A null selector (values 0000–0003) can be loaded into DS or ES without a protection exception. Any memory reference using such a segment register value will cause a #GP(0) exception but will not result in a memory reference. The saved segment register value will be null.

Following is a list of checks and actions taken when loading the DS or ES registers:

```
If selector is non-null then:
  Selector index must be within its descriptor table limits else
   #GP(selector)
  Examine descriptor AR byte:

    Data segment or readable non-conforming code segment
      Descriptor DPL ≥ CPL else #GP (selector)
      Descriptor DPL ≥ selector RPL else #GP (selector)

    Readable conforming code segment
      No DPL, RPL, or CPL checks

    Else #GP (selector)

  Segment must be present else #NP (selector)
  Load registers from operand
  Load segment register descriptor cache

If selector is null then:
  Load registers from operand
  Mark segment register cache as invalid
```

PROTECTED MODE EXCEPTIONS

#GP or #NP, as indicated in the list above. #GP(0) or #SS(0) if operand lies outside segment limit. #UD if the source operand is a register.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for operand at offset 0FFFFH or 0FFFDH. #UD if the source operand is a register.

LEA—Load Effective Address Offset

Opcode	Instruction	Clocks	Description
8D /r	LEA *rw,m*	3	Calculate EA offset given by *m*, place in *rw*

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The effective address (offset part) of the second operand is placed in the first (register) operand.

PROTECTED MODE EXCEPTIONS

#UD if second operand is a register.

REAL ADDRESS MODE EXCEPTIONS

#UD if second operand is a register.

LEAVE—High Level Procedure Exit

Opcode	Instruction	Clocks	Description
C9	LEAVE	5	Set SP to BP, then POP BP

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

LEAVE is the complementary operation to ENTER; it reverses the effects of that instruction. By copying BP to SP, LEAVE releases the stack space used by a procedure for its dynamics and display. The old frame pointer is now popped into BP, restoring the caller's frame, and a subsequent RET instruction will follow the back-link and remove any arguments pushed on the stack for the exiting procedure.

PROTECTED MODE EXCEPTIONS

#SS(0) if BP does not point to a location within the current stack segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 01 /2	LGDT *m*	11	Load *m* into Global Descriptor Table reg
0F 01 /3	LIDT *m*	12	Load *m* into Interrupt Descriptor Table reg

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The Global or the Interrupt Descriptor Table register is loaded from the six bytes of memory pointed to by the effective address operand (see Figure 10-3). The LIMIT field of the descriptor table register loads from the first word; the next three bytes go to the BASE field of the register; the last byte is ignored.

LGDT and LIDT appear in operating systems software; they are not used in application programs. These are the only instructions that directly load a physical memory address in 80C286 protected mode.

PROTECTED MODE EXCEPTIONS

#GP(0) if the current privilege level is not 0.

#UD if source operand is a register.

#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.

REAL ADDRESS MODE EXCEPTIONS

These instructions are valid in Real Address mode to allow the power-up initialization for Protected mode.

Interrupt 13 for a word operand at offset 0FFFFH. #UD if source operand is a register.

LLDT—Load Local Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 00 /2	LLDT *ew*	17,*mem*=19	Load selector *ew* into Local Descriptor Table register

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The word operand (memory or register) to LLDT should contain a selector pointing to the Global Descriptor Table. The GDT entry should be a Local Descriptor Table descriptor. If so, then the Local Descriptor Table register is loaded from the entry. The descriptor cache entries for DS, ES, SS, and CS are not affected. The LDT field in the TSS is not changed.

The selector operand is allowed to be zero. In that case, the Local Descriptor Table register is marked invalid. All descriptor references (except by LAR, VERR, VERW or LSL instructions) will cause a #GP fault.

LLDT appears in operating systems software; it does not appear in applications programs.

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 LDT descriptor is not present. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 6; LLDT is not recognized in Real Address Mode.

LMSW—Load Machine Status Word

Opcode	Instruction	Clocks	Description
0F 01 /6	LMSW *ew*	3,*mem*=6	Load EA word into Machine Status Word

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The Machine Status Word is loaded from the source operand. This instruction may be used to switch to protected mode. If so, then it *must* be followed by an intra-segment jump to flush the instruction queue. LMSW will not switch back to Real Address Mode.

LMSW appears only in operating systems software. It does not appear in applications programs.

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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

LOCK—Assert BUS LOCK Signal

Opcode	Instruction	Clocks	Description
F0	LOCK	0	Assert BUSLOCK signal for the next instruction

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

LOCK is a prefix that will cause the BUS LOCK signal of the 80C286 to be asserted for the duration of the instruction that it prefixes. In a multiprocessor environment, this signal should be used to ensure that the 80C286 has exclusive use of any shared memory while BUS LOCK is asserted. The read-modify-write sequence typically used to implement TEST-AND-SET in the 80C286 is the XCHG instruction.

The 80C286 LOCK prefix activates the lock signal for the following instructions: MOVS, INS, and OUTS. XCHG always asserts BUS LOCK regardless of the presence or absence of the LOCK prefix.

PROTECTED MODE EXCEPTIONS

#GP(0) if the current privilege level is bigger (less privileged) than the I/O privilege level.

Other exceptions may be generated by the subsequent (locked) instruction.

REAL ADDRESS MODE EXCEPTIONS

None. Exceptions may still be generated by the subsequent (locked) instruction.

LODS/LODSB/LODSW—Load String Operand

Opcode	Instruction	Clocks	Description
AC	LODS *mb*	5	Load byte [SI] into AL
AD	LODS *mw*	5	Load word [SI] into AX
AC	LODSB	5	Load byte DS:[SI] into AL
AD	LODSW	5	Load word DS:[SI] into AX

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

LODS loads the AL or AX register with the memory byte or word at SI. After the transfer is made, SI is automatically advanced. If the direction flag is 0 (CLD was executed), SI increments; if the direction flag is 1 (STD was executed), SI decrements. SI increments or decrements by 1 if a byte was moved; by 2 if a word was moved.

PROTECTED MODE EXCEPTIONS

#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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

LOOP/LOOPcond—Loop Control with CX Counter

Opcode	Instruction	Clocks	Description
E2 *cb*	LOOP *cb*	8,*noj*=4	DEC CX; jump short if CX≠0
E1 *cb*	LOOPE *cb*	8,*noj*=4	DEC CX; jump short if CX≠0 and equal (ZF=1)
E0 *cb*	LOOPNE *cb*	8,*noj*=4	DEC CX; jump short if CX≠0 and not equal (ZF=0)
E0 *cb*	LOOPNZ *cb*	8,*noj*=4	DEC CX; jump short if CX≠0 and ZF=0
E1 *cb*	LOOPZ *cb*	8,*noj*=4	DEC CX; jump short if CX≠0 and zero (ZF=1)

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

LOOP first decrements the CX register without changing any of the flags. Then, conditions are checked as given in the description above for the form of LOOP being used. If the conditions are met, then an intra-segment jump is made. The destination to LOOP is in the range from 126 (decimal) bytes before the instruction to 127 bytes beyond the instruction.

The LOOP instructions are intended to provide iteration control and to combine loop index management with conditional branching. To use the LOOP instruction you load an unsigned iteration count into CX, 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.

PROTECTED MODE EXCEPTIONS

#GP(0) if the offset jumped to is beyond the limits of the current code segment.

REAL ADDRESS MODE EXCEPTIONS

None

LSL—Load Segment Limit

Opcode	Instruction	Clocks	Description
0F 03 */r*	LSL *rw,ew*	14,*mem*=16	Load: *rw*=Segment Limit, selector *ew*

FLAGS MODIFIED

Zero

FLAGS UNDEFINED

None

OPERATION

If the descriptor denoted by the selector in the second (memory or register) operand is visible at the CPL, a word that consists of the limit field of the descriptor is loaded into the left operand, which must be a register. The value is the limit field for that segment. The zero flag is set if the loading was performed (i.e., if the selector is non-null, the selector index is within the descriptor table limits, the descriptor is a non-conforming segment descriptor with DPL≥CPL, and the descriptor DPL≥selector RPL); the zero flag is cleared otherwise.

The LSL instruction returns only the limit field of segments, task state segments, and local descriptor tables. The interpretation of the limit value depends on the type of segment.

The selector operand's value cannot result in a protection exception.

PROTECTED MODE EXCEPTIONS

#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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 6; LSL is not recognized in Real Address mode.

LTR—Load Task Register

Opcode	Instruction	Clocks	Description
0F 00 /3	LTR *ew*	17,*mem*=19	Load EA word into Task Register

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The Task Register is loaded from the source register or memory location given by the operand. The loaded TSS is marked busy. A task switch operation does not occur.

LTR appears only in operating systems software. It is not used in applications programs.

PROTECTED MODE EXCEPTIONS

#GP for an illegal memory operand effective address in the CS, DS, or ES segments; #SS 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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 6; LTR is not recognized in Real Address mode.

MOV—Move Data

Opcode	Instruction	Clocks	Description
88 /r	MOV *eb,rb*	2,*mem*=3	Move byte register into EA byte
89 /r	MOV *ew,rw*	2,*mem*=3	Move word register into EA word
8A /r	MOV *rb,eb*	2,*mem*=5	Move EA byte into byte register
8B /r	MOV *rw,ew*	2,*mem*=5	Move EA word into word register
8C /0	MOV *ew*,ES	2,*mem*=3	Move ES into EA word
8C /1	MOV *ew*,CS	2,*mem*=3	Move CS into EA word
8C /2	MOV *ew*,SS	2,*mem*=3	Move SS into EA word
8C /3	MOV *ew*,DS	2,*mem*=3	Move DS into EA word
8E /0	MOV ES,*mw*	5,*pm*=19	Move memory word into ES
8E /0	MOV ES,*rw*	2,*pm*=17	Move word register into ES
8E /2	MOV SS,*mw*	5,*pm*=19	Move memory word into SS
8E /2	MOV SS,*rw*	2,*pm*=17	Move word register into SS
8E /3	MOV DS,*mw*	5,*pm*=19	Move memory word into DS
8E /3	MOV DS,*rw*	2,*pm*=17	Move word register into DS
A0 *dw*	MOV AL,*xb*	5	Move byte variable (offset *dw*) into AL
A1 *dw*	MOV AX,*xw*	5	Move word variable (offset *dw*) into AX
A2 *dw*	MOV *xb*,AL	3	Move AL into byte variable (offset *dw*)
A3 *dw*	MOV *xw*,AX	3	Move AX into word register (offset *dw*)
B0+ *rb db*	MOV *rb,db*	2	Move immediate byte into byte register
B8+ *rw dw*	MOV *rw,dw*	2	Move immediate word into word register
C6 /0 *db*	MOV *eb,db*	2,*mem*=3	Move immediate byte into EA byte
C7 /0 *dw*	MOV *ew,dw*	2,*mem*=3	Move immediate word into EA word

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The second operand is copied to the first operand. If the destination operand is a segment register (DS, ES, or SS), then the associated segment register cache is also loaded. The data for the cache 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 a protection exception. Any use of a segment register with a null selector to address memory will cause #GP(0) exception. No memory reference will occur.

Any move into SS will inhibit all interrupts until after the execution of the next instruction.

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)
  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 cache with descriptor
If ES or DS 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 non-conforming 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
Load segment register cache with descriptor
If ES or DS is loaded with a null selector:
  Load segment register with selector
  Clear descriptor valid bit
```

PROTECTED MODE EXCEPTIONS

If a segment register is being loaded, #GP, #SS, and #NP, as described in the listing above.

Otherwise, #GP(0) if the destination is in a non–writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

MOVS/MOVSB/MOVSW—Move Data from String to String

Opcode	Instruction	Clocks	Description
A4	MOVS *mb,mb*	5	Move byte [SI] to ES:[DI]
A5	MOVS *mw,mw*	5	Move word [SI] to ES:[DI]
A4	MOVSB	5	Move byte DS:[SI] to ES:[DI]
A5	MOVSW	5	Move word DS:[SI] to ES:[DI]

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

MOVS copies the byte or word at [SI] to the byte or word at ES:[DI]. The destination operand must be addressable from the ES register; no segment override is possible. A segment override may be used for the source operand.

After the data movement is made, both SI and DI 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 was moved; by 2 if a word 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.

PROTECTED MODE EXCEPTIONS

#GP(0) if the destination is in a non-writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

MUL—Unsigned Multiplication of AL or AX

Opcode	Instruction	Clocks	Description
F6 /4	MUL *eb*	13,*mem*=16	Unsigned multiply (AX = AL × EA byte)
F7 /4	MUL *ew*	21,*mem*=24	Unsigned multiply (DXAX = AX × EA word)

FLAGS MODIFIED

Overflow, carry

FLAGS UNDEFINED

Sign, zero, auxiliary carry, parity

OPERATION

If MUL has a byte operand, then the byte is multiplied by AL, and the result is left in AX. Carry and overflow are set to 0 if AH is 0; they are set to 1 otherwise.

If MUL has a word operand, then the word is multiplied by AX, and the result is left in DX:AX. DX contains the high order 16 bits of the product. Carry and overflow are set to 0 if DX is 0; they are set to 1 otherwise.

PROTECTED MODE EXCEPTIONS

#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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

NEG—Two's Complement Negation

Opcode	Instruction	Clocks	Description
F6 /3	NEG *eb*	2,*mem*=7	Two's complement negate EA byte
F7 /3	NEG *ew*	2,*mem*=7	Two's complement negate EA word

FLAGS MODIFIED

Overflow, sign, zero, auxiliary carry, parity, carry

FLAGS UNDEFINED

None

OPERATION

The two's complement of the register or memory operand replaces the old operand value. Likewise, the operand is subtracted from zero, and the result is placed in the operand.

The carry flag is set to 1 except when the input operand is zero, in which case the carry flag is cleared to 0.

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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

NOP—No OPERATION

Opcode	Instruction	Clocks	Description
90	NOP	3	No OPERATION

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

Performs no operation. NOP is a one-byte filler instruction that takes up space but affects none of the machine context except IP.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

NOT—One's Complement Negation

Opcode	Instruction	Clocks	Description
F6 /2	NOT *eb*	2,*mem*=7	Reverse each bit of EA byte
F7 /2	NOT *ew*	2,*mem*=7	Reverse each bit of EA word

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The operand is inverted; that is, every 1 becomes a 0 and vice versa.

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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

OR—Logical Inclusive OR

Opcode	Instruction	Clocks	Description
08 /r	OR *eb,rb*	2,*mem*=7	Logical-OR byte register into EA byte
09 /r	OR *ew,rw*	2,*mem*=7	Logical-OR word register into EA word
0A /r	OR *rb,eb*	2,*mem*=7	Logical-OR EA byte into byte register
0B /r	OR *rw,ew*	2,*mem*=7	Logical-OR EA word into word register
0C *db*	OR AL,*db*	3	Logical-OR immediate byte into AL
0D *dw*	OR AX,*dw*	3	Logical-OR immediate word into AX
80 /1 *db*	OR *eb,db*	3,*mem*=7	Logical-OR immediate byte into EA byte
81 /1 *dw*	OR *ew,dw*	3,*mem*=7	Logical-OR immediate word into EA word

FLAGS MODIFIED

Overflow = 0, sign, zero, parity, carry = 0

FLAGS UNDEFINED

Auxiliary carry

OPERATION

This instruction computes the inclusive OR of the two operands. Each bit of the result is 0 if both corresponding bits of the operands are 0; each bit is 1 otherwise. The result is placed in the first operand.

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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

OUT—Output to Port

Opcode	Instruction	Clocks	Description
E6 *db*	OUT *db*,AL	3	Output byte AL to immediate port number *db*
E7 *db*	OUT *db*,AX	3	Output word AX to immediate port number *db*
EE	OUT DX,AL	3	Output byte AL to port number DX
EF	OUT DX,AX	3	Output word AX to port number DX

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

OUT transfers a data byte or data word from the register (AL or AX) given as the second operand to the output port numbered by the first operand. You can output to any port from 0-65535 by placing the port number in the DX register then using an OUT instruction with DX as the first operand. If the instruction contains an 8-bit port ID, that value is zero-extended to 16 bits.

I/O port addresses 00F8H through 00FFH are reserved; these addresses should not be used.

PROTECTED MODE EXCEPTIONS

#GP(0) if the current privilege level is bigger (has less privilege) than IOPL, which is the privilege level found in the flags register.

REAL ADDRESS MODE EXCEPTIONS

None

OUTS/OUTSB/OUTSW—Output String to Port

Opcode	Instruction	Clocks	Description
6E	OUTS DX,*eb*	5	Output byte [SI] to port number DX
6F	OUTS DX,*ew*	5	Output word [SI] to port number DX
6E	OUTSB	5	Output byte DS:[SI] to port number DX
6F	OUTSW	5	Output word DS:[SI] to port number DX

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

OUTS transfers data from the memory byte or word at SI to the output port numbered by the DX register.

OUTS does not allow the specification of the port number as an immediate value. The port must be addressed through the DX register.

After the transfer is made, SI is automatically advanced. If the direction flag is 0 (CLD was executed), SI increments; if the direction flag is 1 (STD was executed), SI decrements. SI increments or decrements by 1 if a byte was moved; by 2 if a word was moved.

OUTS can be preceded by the REP prefix for block output of CX bytes or words. Refer to the REP instruction for details of this operation.

I/O port addresses 00F8H through 00FFH are reserved; these addresses should not be used.

NOTE: Not all output devices can handle the rate at which this instruction transfers data.

PROTECTED MODE EXCEPTIONS

#GP(0) if CPL > IOPL. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

POP—Pop a Word from the Stack

Opcode	Instruction	Clocks	Description
1F	POP DS	5,*pm*=20	Pop top of stack into DS
07	POP ES	5,*pm*=20	Pop top of stack into ES
17	POP SS	5,*pm*=20	Pop top of stack into SS
8F /0	POP *mw*	5	Pop top of stack into memory word
58+ *rw*	POP *rw*	5	Pop top of stack into word register

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The word on the top of the 80C286 stack, addressed by SS:SP, replaces the previous contents of the memory, register, or segment register operand. The stack pointer SP is incremented by 2 to point to the new top of stack.

If the destination operand is another segment register (DS, ES, 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 loaded into the DS or ES register without causing a protection exception. Attempts to reference memory using a segment register with a null value will cause #GP(0) exception. No memory reference will occur. The saved value of the segment register will be null.

A POP SS instruction will inhibit all interrupts, including NMI, until after the execution of the next instruction. This permits a POP SP instruction to be performed first.

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)
 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 cache with descriptor

If ES or DS is loaded with non-null  selector:
 AR byte must indicate data or readable code segment else #GP (selector)
 If data or non-conforming 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
  Load segment register cache with descriptor
If ES or DS is loaded with a  null selector:
  Load segment register with selector
  Clear valid bit in cache
```

PROTECTED MODE EXCEPTIONS

If a segment register is being loaded, #GP, #SS, and #NP, as described in the listing above.

Otherwise, #SS(0) if the current top of stack is not within the stack segment.

#GP(0) if the destination is in a non-writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

POPA—Pop All General Registers

Opcode	Instruction	Clocks	Description
61	POPA	19	Pop in order: DI,SI,BP,SP,BX,DX,CX,AX

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

POPA pops the eight general registers given in the description above, except that 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.

PROTECTED MODE EXCEPTIONS

#SS(0) if the starting or ending stack address is not within the stack segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

POPF—Pop from Stack into the Flags Register

Opcode	Instruction	Clocks	Description
9D	POPF	5	Pop top of stack into flags register

FLAGS MODIFIED

Entire flags register is popped from stack

FLAGS UNDEFINED

None

OPERATION

The top of the 80C286 stack, pointed to by SS:SP, is copied into the 80C286 flags register. The stack pointer SP is incremented by 2 to point to the new top of stack. The flags, from the top bit (bit 15) to the bottom (bit 0), are as follows: undefined, nested task, I/O privilege level (2 bits), overflow, direction, interrupts enabled, trap, sign, zero, undefined, auxiliary carry, undefined, parity, undefined, and carry.

The I/O privilege level will be altered only when executing at privilege level 0. The interrupt enable flag will be altered only when executing at a level at least as privileged as the I/O privilege level. If you execute a POPF instruction with insufficient privilege, there will be no exception nor will the privileged bits be changed.

PROTECTED MODE EXCEPTIONS

#SS(0) if the top of stack is not within the stack segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at 0FFFFH.

In real mode the NT and IOPL bits will not be modified.

PUSH—Push a Word onto the Stack

Opcode	Instruction	Clocks	Description
06	PUSH ES	3	Push ES
0E	PUSH CS	3	Push CS
16	PUSH SS	3	Push SS
1E	PUSH DS	3	Push DS
50+ *rw*	PUSH *rw*	3	Push word register
FF /6	PUSH *mw*	5	Push memory word
68 *dw*	PUSH *dw*	3	Push immediate word
6A *db*	PUSH *db*	3	Push immediate sign-extended byte

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The stack pointer SP is decremented by 2, and the operand is placed on the new top of stack, which is pointed to by SS:SP.

The 80C286 PUSH SP instruction pushes the value of SP as it existed before the instruction. This differs from the 8086, which pushes the new (decremented by 2) value.

PROTECTED MODE EXCEPTIONS

#SS(0) if the new value of SP is outside the stack segment limit.

#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.

REAL ADDRESS MODE EXCEPTIONS

None; the 80C286 will shut down if SP = 1 due to lack of stack space.

PUSHA—Push All General Registers

Opcode	Instruction	Clocks	Description
60	PUSHA	17	Push in order: AX,CX,DX,BX, original SP,BP,SI,DI

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

PUSHA saves the registers noted above on the 80C286 stack. The stack pointer SP is decremented by 16 to hold the eight-word values. Since the registers are pushed onto the stack in the order in which they were given, they will appear in the 16 new stack bytes in the reverse order. The last register pushed is DI.

PROTECTED MODE EXCEPTIONS

#SS(0) if the starting or ending address is outside the stack segment limit.

REAL ADDRESS MODE EXCEPTIONS

The 80C286 will shut down if SP = 1, 3, or 5 before executing PUSHA. If SP = 7, 9, 11, 13, or 15, Exception 13 will occur.

PUSHF—Push Flags Register onto the Stack

Opcode	Instruction	Clocks	Description
9C	PUSHF	3	Push flags register

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The stack pointer SP is decremented by 2, and the 80C286 flags register is copied to the new top of stack, which is pointed to by SS:SP. The flags, from the top bit (15) to the bottom bit (0), are as follows: undefined, nested task, I/O privilege level (2 bits), overflow, direction, interrupts enabled, trap, sign, zero, undefined, auxiliary carry, undefined, parity, undefined, and carry.

PROTECTED MODE EXCEPTIONS

#SS(0) if the new value of SP is outside the stack segment limit.

REAL ADDRESS MODE EXCEPTIONS

None; the 80C286 will shut down if SP = 1 due to lack of stack space.

RCL/RCR/ROL/ROR—Rotate Instructions

Opcode			Instruction	Clocks-N*	Description
D0	/2		RCL *eb*,1	2,*mem*=7	Rotate 9-bits (CF, EA byte) left once
D2	/2		RCL *eb*,CL*	5,*mem*=8	Rotate 9-bits (CF, EA byte) left CL times
C0	/2	*db*	RCL *eb,db**	5,*mem*=8	Rotate 9-bits (CF, EA byte) left *db* times
D1	/2		RCL *ew*,1	2,*mem*=7	Rotate 17-bits (CF, EA word) left once
D3	/2		RCL *ew*,CL*	5,*mem*=8	Rotate 17-bits (CF, EA word) left CL times
C1	/2	*db*	RCL *ew,db**	5,*mem*=8	Rotate 17-bits (CF, EA word) left *db* times
D0	/3		RCR *eb*,1	2,*mem*=7	Rotate 9-bits (CF, EA byte) right once
D2	/3		RCR *eb*,CL	5,*mem*=8	Rotate 9-bits (CF, EA byte) right CL times
C0	/3	*db*	RCR *eb,db**	5,*mem*=8	Rotate 9-bits (CF, EA byte) right *db* times
D1	/3		RCR *ew*,1	2,*mem*=7	Rotate 17-bits (CF, EA word) right once
D3	/3		RCR *ew*,CL*	5,*mem*=8	Rotate 17-bits (CF, EA word) right CL times
C1	/3	*db*	RCR *ew,db**	5,*mem*=8	Rotate 17-bits (CF, EA word) right *db* times
D0	/0		ROL *eb*,1	2,*mem*=7	Rotate 8-bit EA byte left once
D2	/0		ROL *eb*,CL*	5,*mem*=8	Rotate 8-bit EA byte left CL times
C0	/0	*db*	ROL *eb,db**	5,*mem*=8	Rotate 8-bit EA byte left *db* times
D1	/0		ROL *ew*,1	2,*mem*=7	Rotate 16-bit EA word left once
D3	/0		ROL *ew*,CL*	5,*mem*=8	Rotate 16-bit EA word left CL times
C1	/0	*db*	ROL *ew,db**	5,*mem*=8	Rotate 16-bit EA word left *db* times
D0	/1		ROR *eb*,1	2,*mem*=7	Rotate 8-bit EA byte right once
D2	/1		ROR *eb*,CL*	5,*mem*=8	Rotate 8-bit EA byte right CL times
C0	/1	*db*	ROR *eb,db**	5,*mem*=8	Rotate 8-bit EA byte right *db* times
D1	/1		ROR *ew*,1	2,*mem*=7	Rotate 16-bit EA word right once
D3	/1		ROR *ew*,CL*	5,*mem*=8	Rotate 16-bit EA word right CL times
C1	/1	*db*	ROR *ew,db**	5,*mem*=8	Rotate 16-bit EA word right *db* times

* Add 1 clock to the times shown for each rotate made.

FLAGS MODIFIED

Overflow (only for single rotates), carry

FLAGS UNDEFINED

Overflow for multi-bit rotates

OPERATION

Each rotate instruction shifts the bits of the register or memory operand given. The left rotate instructions shift all of the bits upward, except for the top bit, which comes back around to the bottom. The right rotate instructions do the reverse: the bits shift downward, with the bottom bit coming around to 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; nonetheless, 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 execution time, the 80C286 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 rotate counts.

The overflow flag is set only for the single-rotate (second operand = 1) forms of the instructions. The OF bit is set to be accurate if a shift of length 1 is done. Since it is undefined for all other values, including a zero shift, it can always be set for the count-of-1 case regardless of the actual count. 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. Neither flag bit is modified when the count value is zero.

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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

REP/REPE/REPNE—Repeat Following String Operation

Opcode	Instruction	Clocks*	Description
F3 6C	REP INS *eb*,DX	5+4 *CX	Input CX bytes from port DX into ES:[DI]
F3 6D	REP INS *ew*,DX	5+4 *CX	Input CX words from port DX into ES:[DI]
F3 6C	REP INSB	5+4 *CX	Input CX bytes from port DX into ES:[DI]
F3 6D	REP INSW	5+4 *CX	Input CX words from port DX into ES:[DI]
F3 A4	REP MOVS *mb,mb*	5+4 *CX	Move CX bytes from [SI] to ES:[DI]
F3 A5	REP MOVS *mw,mw*	5+4 *CX	Move CX words from [SI] to ES:[DI]
F3 A4	REP MOVSB	5+4 *CX	Move CX bytes from DS:[SI] to ES:[DI]
F3 A5	REP MOVSW	5+4 *CX	Move CX words from DS:[SI] to ES:[DI]
F3 6E	REP OUTS DX, *eb*	5+4 *CX	Output CX bytes from [SI] to port DX
F3 6F	REP OUTS DX, *ew*	5+4 *CX	Output CX words from [SI] to port DX
F3 6E	REP OUTSB	5+4 *CX	Output CX bytes from DS:[SI] to port DX
F3 6F	REP OUTSW	5+4 *CX	Output CX words from DS:[SI] to port DX
F3 AA	REP STOS *mb*	4+3 *CX	Fill CX bytes at ES:[DI] with AL
F3 AB	REP STOS *mw*	4+3 *CX	Fill CX words at ES:[DI] with AX
F3 AA	REP STOSB	4+3 *CX	Fill CX bytes at ES:[DI] with AL
F3 AB	REP STOSW	4+3 *CX	Fill CX words at ES:[DI] with AX
F3 A6	REPE CMPS *mb,mb*	5+9 *N	Find nonmatching bytes in ES:[DI] and [SI]
F3 A7	REPE CMPS *mw,mw*	5+9 *N	Find nonmatching words in ES:[DI] and [SI]
F3 A6	REPE CMPSB	5+9 *N	Find nonmatching bytes in ES:[DI] and DS:[SI]
F3 A7	REPE CMPSW	5+9 *N	Find nonmatching words in ES:[DI] and DS:[SI]
F3 AE	REPE SCAS *mb*	5+8 *N	Find non-AL byte starting at ES:[DI]
F3 AF	REPE SCAS *mw*	5+8 *N	Find non-AX word starting at ES:[DI]
F3 AE	REPE SCASB	5+8 *N	Find non-AL byte starting at ES:[DI]
F3 AF	REPE SCASW	5+8 *N	Find non-AX word starting at ES:[DI]
F2 A6	REPNE CMPS *mb,mb*	5+9 *N	Find matching bytes in ES:[DI] and [SI]
F2 A7	REPNE CMPS *mw,mw*	5+9 *N	Find matching words in ES:[DI] and [SI]
F2 A6	REPNE CMPSB	5+9 *N	Find matching bytes in ES:[DI] and DS:[SI]
F2 A7	REPNE CMPSW	5+9 *N	Find matching words in ES:[DI] and DS:[SI]
F2 AE	REPNE SCAS *mb*	5+8 *N	Find AL, starting at ES:[DI]
F2 AF	REPNE SCAS *mw*	5+8 *N	Find AX, starting at ES:[DI]
F2 AE	REPNE SCASB	5+8 *N	Find AL, starting at ES:[DI]
F2 AF	REPNE SCASW	5+8 *N	Find AX, starting at ES:[DI]

*N denotes the number of iterations actually executed.

FLAGS MODIFIED

By CMPS and SCAS, none by REP

FLAGS UNDEFINED

None

OPERATION

REP, REPE, and REPNE are prefix operations. These prefixes cause the string instruction that follows to be repeated CX times or (for REPE and REPNE) until the indicated condition in the zero flag is no longer met. Thus, REPE stands for Repeat While Equal, REPNE for Repeat While Not Equal.

The REP prefixes make sense only in the contexts listed above. They cannot be applied to anything other than string operations.

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 a LOOP construct.

The precise action for each iteration is as follows:

1. Check the CX register. If it is zero, exit the iteration and move to the next instruction.
2. Acknowledge any pending interrupts.
3. Perform the string operation once.
4. Decrement CX by 1; no flags are modified.
5. If the string operation is SCAS or CMPS, check the zero flag. If the repeat condition does not hold, then exit the iteration and move to the next instruction. Exit if the prefix is REPE and ZF = 0 (the last comparison was not equal), or if the prefix is REPNE and ZF = 1 (the last comparison was equal).
6. Go to step 1 for the next iteration.

As defined by the individual string-ops, the direction of movement through the block is determined by the direction flag. If the direction flag is 1 (STD was executed), SI and/or DI start at the end of the block and move backward; if the direction flag is 0 (CLD was executed), SI and/or DI start at the beginning of the block and move forward.

For repeated SCAS and CMPS operations the repeat can be exited for one of two different reasons: the CX count can be exhausted or the zero flag can fail the repeat condition. Your code will probably want to distinguish between the two cases. It can do so via either the JCXZ instruction or the conditional jumps that test the zero flag (JZ, JNZ, JE, and JNE).

Note: Not all input/output ports can handle the rate at which the repeated I/O instructions execute.

PROTECTED MODE EXCEPTIONS

None by REP; exceptions can be generated when the string-op is executed.

REAL ADDRESS MODE EXCEPTIONS

None by REP; exceptions can be generated when the string-op is executed.

RET—Return from Procedure

Opcode	Instruction	Clocks*	Description
CB	RET	15,*pm*=25	Return to far caller, same privilege
CB	RET	55	Return, lesser privilege, switch stacks
C3	RET	11	Return to near caller, same privilege
CA *dw*	RET *dw*	15,*pm*=25	RET (far), same privilege, pop *dw* bytes
CA *dw*	RET *dw*	55	RET (far), lesser privilege, pop *dw* bytes
C2 *dw*	RET *dw*	11	RET (near), same privilege, pop *dw* bytes pushed before Call

* Add 1 clock to each byte in the next instruction executed.

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

RET transfers control to a return address located on the stack. The address is usually placed on the stack by a CALL instruction; in that case, the return is made to the instruction that follows the CALL.

There is an optional numeric parameter to RET. It gives the number of stack bytes to be released after the return address is popped. These bytes are typically used as input parameters to the procedure called.

For the intra-segment return, the address on the stack is a 2-byte quantity popped into IP. The CS register is unchanged.

For the inter-segment return, the address on the stack is a 4-byte-long pointer. The offset is popped first, followed by the selector. In Real Address mode, CS and IP are directly loaded.

In protected mode, an inter-segment return causes the processor to consult the descriptor addressed by the return selector. The AR byte of the descriptor must indicate a code segment of equal or less privilege (of 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 and ES segment registers may be set to zero by the inter-segment RET instruction. If these registers refer to segments which cannot be used by the new privilege level, they are set to zero to prevent unauthorized access.

The following list of checks and actions describes the Protected mode inter-segment return in detail.

```
Inter-segment RET:
 Second word on stack must be within stack limits else #SS(0)
 Return selector RPL must be ≥ CPL else #GP (return selector)
 If return selector RPL = CPL then
```

```
  RETURN TO 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)
  If conforming then code segment DPL must be ≤ CPL else #GP (selector)
  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)
  Load CS:IP from stack
  Load CS-cache with descriptor
  Increment SP by 4 plus the immediate offset if it exists
Else
  RETURN TO OUTER PRIVILEGE LEVEL:
  Top (8 + immediate) bytes on stack must be within stack limits else #SS(0)
  Examine return CS selector (at SP + 2) 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)
   If conforming then code segment DPL must be ≤ return selector RPL else #GP
     (selector)
   Segment must be PRESENT else #NP (selector)
  Examine return SS selector (at SP + 6 + imm) 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 #SS (selector)
  IP must be in code segment limit else # GP(0)
  Set CPL to the RPL of the return CS selector
  Load CS:IP from stack
  Set CS RPL to CPL
  Increment SP by 4 plus the immediate offset if it exists
  Load SS:SP from stack
  Load the CS-cache with the return CS descriptor
  Load the SS-cache with the return SS descriptor
  For each of ES and DS:
   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
```

PROTECTED MODE EXCEPTIONS

#GP, #NP, or #SS, as described in the above listing.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 if the stack pop wraps around from 0FFFFH to 0.

SAHF—Store AH into Flags

Opcode	Instruction	Clocks	Description
9E	SAHF	2	Store AH into flags SF ZF xx AF xx PF xx CF

FLAGS MODIFIED

Sign, zero, auxiliary carry, parity, carry

FLAGS UNDEFINED

None

OPERATION

The flags listed above are loaded with values from the AH register, from bits 7, 6, 4, 2, and 0, respectively.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

SAL/SAR/SHL/SHR—Shift Instructions

Opcode			Instruction	Clocks-N*	Description
D0	/4		SAL *eb*,1	2,mem = 7	Multiply EA byte by 2, once
D2	/4		SAL *eb*,CL	5,mem = 8	Multiply EA byte by 2, CL times
C0	/4	*db*	SAL *eb,db*	5,mem = 8	Multiply EA byte by 2, *db* times
D1	/4		SAL *ew*,1	2,mem = 7	Multiply EA word by 2, once
D3	/4		SAL *ew*,CL	5,*mem* = 8	Multiply EA word by 2, CL times
C1	/4	*db*	SAL *ew,db*	5,*mem* = 8	Multiply EA word by 2, *db* times
D0	/7		SAR *eb*,1	2,*mem* = 7	Signed divide EA byte by 2, once
D2	/7		SAR *eb*,CL	5,*mem* = 8	Signed divide EA byte by 2, CL times
C0	/7	*db*	SAR *eb,db*	5,*mem* = 8	Signed divide EA byte by 2, *db* times
D1	/7		SAR *ew*,1	2,*mem* = 7	Signed divide EA word by 2, once
D3	/7		SAR *ew*,CL	5,*mem* = 8	Signed divide EA word by 2, CL times
C1	/7	*db*	SAR *ew,db*	5,*mem* = 8	Signed divide EA word by 2, *db* times
D0	/5		SHR *eb*,1	2,*mem* = 7	Unsigned divide EA byte by 2, once
D2	/5		SHR *eb*,CL	5,*mem* = 8	Unsigned divide EA byte by 2, CL times
C0	/5	*db*	SHR *eb,db*	5,*mem* = 8	Unsigned divide EA byte by 2, *db* times
D1	/5		SHR *ew*,1	2,*mem* = 7	Unsigned divide EA word by 2, once
D3	/5		SHR *ew*,CL	5,*mem* = 8	Unsigned divide EA word by 2, CL times
C1	/5		SHR *ew,db*	5,*mem* = 8	Unsigned divide EA word by 2, CL *db* times

*Add 1 clock to the times shown for each shift performed.

FLAGS MODIFIED

Overflow (only for single-shift form), carry, zero, parity, sign

FLAGS UNDEFINED

Auxiliary carry; also overflow for multibit shifts (only).

OPERATION

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: 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 80C286 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 8 bits of the shift count.

The overflow flag is set only if the single-shift forms of the instructions are used. For left shifts, it is set to 0 if the high bit of the answer is the same as the result carry flag (i.e., the top two bits of the original operand were the same); it is set to 1 if they are different. For SAR it is set to 0 for all single shifts. For SHR, it is set to the high-order bit of the original operand. Neither flag bit is modified when the count value is zero.

PROTECTED MODE EXCEPTIONS

#GP(0) if the operand is in a non-writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

SBB—Integer Subtraction With Borrow

Opcode	Instruction	Clocks	Description
18 /r	SBB *eb,rb*	2,*mem*=7	Subtract with borrow byte register from EA byte
19 /r	SBB *ew,rw*	2,*mem*=7	Subtract with borrow word register from EA word
1A /r	SBB *rb,eb*	2,*mem*=7	Subtract with borrow EA byte from byte register
1B /r	SBB *rw,ew*	2,*mem*=7	Subtract with borrow EA word from word register
1C *db*	SBB AL,*db*	3	Subtract with borrow imm. byte from AL
1D *dw*	SBB AX,*dw*	3	Subtract with borrow imm. word from AX
80 /3 *db*	SBB *eb,db*	3,*mem*=7	Subtract with borrow imm. byte from EA byte
81 /3 *dw*	SBB *ew,dw*	3,*mem*=7	Subtract with borrow imm. word from EA word
83 /3 *db*	SBB *ew,db*	3,*mem*=7	Subtract with borrow imm. byte from EA word

FLAGS MODIFIED

Overflow, sign, zero, auxiliary carry, parity, carry

FLAGS UNDEFINED

None

OPERATION

The second operand is added to the carry flag and the result is subtracted from the first operand. The first operand is replaced with the result of the subtraction, and the flags are set accordingly.

When a byte-immediate value is subtracted from a word operand, the immediate value is first sign-extended.

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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

SCAS/SCASB/SCASW—Compare String Data

Opcode	Instruction	Clocks	Description
AE	SCAS *mb*	7	Compare bytes AL – ES:[DI], advance DI
AF	SCAS *mw*	7	Compare words AX – ES:[DI], advance DI
AE	SCASB	7	Compare bytes AL – ES:[DI], advance DI
AF	SCASW	7	Compare words AX – ES:[DI], advance DI

FLAGS MODIFIED

Overflow, sign, zero, auxiliary carry, parity, carry

FLAGS UNDEFINED

None

OPERATION

SCAS subtracts the memory byte or word at ES:DI from the AL or AX register. The result is discarded; only the flags are set. The operand must be addressable from the ES register; no segment override is possible.

After the comparison is made, DI is automatically advanced. If the direction flag is 0 (CLD was executed), DI increments; if the direction flag is 1 (STD was executed), DI decrements. DI increments or decrements by 1 if bytes were compared; by 2 if words were compared.

SCAS can be preceded by the REPE or REPNE prefix for a block search of CX bytes or words. Refer to the REP instruction for details of this operation.

PROTECTED MODE EXCEPTIONS

#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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

SGDT/SIDT—Store Global/Interrupt Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 01 /0	SGDT *m*	11	Store Global Descriptor Table register to *m*
0F 01 /1	SIDT *m*	12	Store Interrupt Descriptor Table register to *m*

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The contents of the descriptor table register are copied to six bytes of memory indicated by the operand. The LIMIT field of the register goes to the first word at the effective address; the next three bytes get the BASE field of the register; and the last byte is undefined.

SGDT and SIDT appear only in operating systems software; they are not used in applications programs.

PROTECTED MODE EXCEPTIONS

#UD if the destination operand is a register. #GP(0) if the destination is in a non-writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

These instructions are valid in Real Address mode to facilitate power-up or to reset initialization prior to entering Protected mode.

#UD if the destination operand is a register. Interrupt 13 for a word operand at offset 0FFFFH.

SLDT—Store Local Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 00 /0	SLDT *ew*	2,*mem*=3	Store Local Descriptor Table register to EA word

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The Local Descriptor Table register is stored in the 2-byte register or memory location indicated by the effective address operand. This register is a selector that points into the Global Descriptor Table.

SLDT appears only in operating systems software. It is not used in applications programs.

PROTECTED MODE EXCEPTIONS

#GP(0) if the destination is in a non-writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 6; SLDT is not recognized in Real Address mode.

SMSW—Store Machine Status Word

Opcode	Instruction	Clocks	Description
0F 01 /4	SMSW *ew*	2,*mem*=3	Store Machine Status Word to EA word

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The Machine Status Word is stored in the 2-byte register or memory location indicated by the effective address operand.

PROTECTED MODE EXCEPTIONS

#GP(0) if the destination is in a non-writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

STC—Set Carry Flag

Opcode	Instruction	Clocks	Description
F9	STC	2	Set carry flag

FLAGS MODIFIED

Carry = 1

FLAGS UNDEFINED

None

OPERATION

The carry flag is set to 1.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

STD—Set Direction Flag

Opcode	Instruction	Clocks	Description
FD	STD	2	Set direction flag so SI and DI will decrement

FLAGS MODIFIED

Direction = 1

FLAGS UNDEFINED

None

OPERATION

The direction flag is set to 1. This causes all subsequent string operations to decrement the index registers (SI and/or DI) on which they operate.

PROTECTED MODE EXCEPTIONS

None

REAL ADDRESS MODE EXCEPTIONS

None

STI—Set Interrupt Enable Flag

Opcode	Instruction	Clocks	Description
FB	STI	2	Set interrupt enable flag, interrupts enabled

FLAGS MODIFIED

Interrupt = 1 (enabled)

FLAGS UNDEFINED

None

OPERATION

The interrupts-enabled flag is set to 1. The 80C286 will now respond to external interrupts after executing the STI instruction.

PROTECTED MODE EXCEPTIONS

#GP(0) if the current privilege level is bigger (has less privilege) than the I/O privilege level.

REAL ADDRESS MODE EXCEPTIONS

None

STOS/STOSB/STOSW—Store String Data

Opcode	Instruction	Clocks	Description
AA	STOS *mb*	3	Store AL to byte ES:[DI], advance DI
AB	STOS *mw*	3	Store AX to word ES:[DI], advance DI
AA	STOSB	3	Store AL to byte ES:[DI], advance DI
AB	STOSW	3	Store AX to word ES:[DI], advance DI

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

STOS transfers the contents the AL or AX register to the memory byte or word at ES:DI. The operand must be addressable from the ES register; no segment override is possible.

After the transfer is made, DI is automatically advanced. If the direction flag is 0 (CLD was executed), DI increments; if the direction flag is 1 (STD was executed), DI decrements. DI increments or decrements by 1 if a byte was moved; by 2 if a word was moved.

STOS can be preceded by the REP prefix for a block fill of CX bytes or words. Refer to the REP instruction for details of this operation.

PROTECTED MODE EXCEPTIONS

#GP(0) if the destination is in a non-writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

STR—Store Task Register

Opcode	Instruction	Clocks	Description
0F 00 /1	STR *ew*	2,*mem*=3	Store Task Register to EA word

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The contents of the Task Register are copied to the 2-byte register or memory location indicated by the effective address operand.

PROTECTED MODE EXCEPTIONS

#GP(0) if the destination is in a non-writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 6; STR is not recognized in Real Address mode.

SUB—Integer Subtraction

Opcode	Instruction	Clocks	Description
28 /r	SUB *eb,rb*	2,*mem*=7	Subtract byte register from EA byte
29 /r	SUB *ew,rw*	2,*mem*=7	Subtract word register from EA word
2A /r	SUB *rb,eb*	2,*mem*=7	Subtract EA byte from byte register
2B /r	SUB *rw,ew*	2,*mem*=7	Subtract EA word from word register
2C *db*	SUB AL,*db*	3	Subtract immediate byte from AL
2D *dw*	SUB AX,*dw*	3	Subtract immediate word from AX
80 /5 *db*	SUB *eb,db*	3,*mem*=7	Subtract immediate byte from EA byte
81 /5 *dw*	SUB *ew,dw*	3,*mem*=7	Subtract immediate word from EA word
83 /5 *db*	SUB *ew,db*	3,*mem*=7	Subtract immediate byte from EA word

FLAGS MODIFIED

Overflow, sign, zero, auxiliary carry, parity, carry

FLAGS UNDEFINED

None

OPERATION

The second operand is subtracted from the first operand, and the first operand is replaced with the result.

When a byte-immediate value is subtracted from a word operand, the immediate value is first sign-extended.

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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

TEST—Logical Compare

Opcode	Instruction	Clocks	Description
84 /r	TEST *eb,rb*	2,*mem*=6	AND byte register into EA byte for flags only
84 /r	TEST *rb,eb*	2,*mem*=6	AND EA byte into byte register for flags only
85 /r	TEST *ew,rw*	2,*mem*=6	AND word register into EA word for flags only
85 /r	TEST *rw,ew*	2,*mem*=6	AND EA word into word register for flags only
A8 *db*	TEST AL,*db*	3	AND immediate byte into AL for flags only
A9 *dw*	TEST AX,*dw*	3	AND immediate word into AX for flags only
F6 /0 *db*	TEST *eb,db*	3,*mem*=6	AND immediate byte into EA byte for flags only
F7 /0 *dw*	TEST *ew,dw*	3,*mem*=6	AND immediate word into EA word for flags only

FLAGS MODIFIED

Overflow = 0, sign, zero, parity, carry = 0

FLAGS UNDEFINED

Auxiliary carry

OPERATION

TEST computes the bit-wise logical AND of the two operands given. Each bit of the result is 1 if both of the corresponding bits of the operands are 1; each bit is 0 otherwise. The result of the operation is discarded; only the flags are modified.

PROTECTED MODE EXCEPTIONS

#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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

VERR,VERW—Verify a Segment for Reading or Writing

Opcode	Instruction	Clocks	Description
0F 00 /4	VERR *ew*	14,*mem*=16	Set ZF=1 if seg. can be read, selector *ew*
0F 00 /5	VERW *ew*	14,*mem*=16	Set ZF=1 if seg. can be written, selector *ew*

FLAGS MODIFIED

Zero

FLAGS UNDEFINED

None

OPERATION

VERR and VERW expect the 2-byte register or memory operand to contain the value of a selector. The instructions determine whether the segment denoted by the selector is reachable from the current privilege level; the instructions also determine whether it is readable or writable. If the segment is determined to be accessible, the zero flag is set to 1; if the segment is not accessible, it is set to 0. To set ZF, the following conditions must be met:

1. The selector must denote a descriptor within the bounds of the table (GDT or LDT); that is, the selector must be defined.
2. The selector must denote the descriptor of a code or data segment.
3. If the instruction is VERR, the segment must be readable. If the instruction is VERW, the segment must be a writable data segment.
4. 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 or ES 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. This enables the software to anticipate possible segment access problems.

PROTECTED MODE EXCEPTIONS

The only faults that can occur are those generated by illegally addressing the memory operand which 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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 6; VERR and VERW are not recognized in Real Address mode.

WAIT—Wait Until $\overline{\text{BUSY}}$ Pin Is Inactive (High)

Opcode	Instruction	Clocks	Description
9B	WAIT	3	Wait until $\overline{\text{BUSY}}$ pin is inactive (High)

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

WAIT suspends execution of 80C286 instructions until the $\overline{\text{BUSY}}$ pin is inactive (High). The $\overline{\text{BUSY}}$ pin is driven by the AMD 80C287 math coprocessor. WAIT is issued to ensure that the numeric instruction being executed is complete, and to check for a possible numeric fault (see below).

PROTECTED MODE EXCEPTIONS

#NM if task switch flag in MSW is set. #MF if the AMD 80C287 coprocessor has detected an unmasked numeric error.

REAL ADDRESS MODE EXCEPTIONS

Same as Protected mode.

XCHG—Exchange Memory/Register with Register

Opcode	Instruction	Clocks	Description
86 */r*	XCHG *eb,rb*	3,*mem*=5	Exchange byte register with EA byte
86 */r*	XCHG *rb,eb*	3,*mem*=5	Exchange EA byte with byte register
87 */r*	XCHG *ew,rw*	3,*mem*=5	Exchange word register with EA word
87 */r*	XCHG *rw,ew*	3,*mem*=5	Exchange EA word with word register
90+*rw*	XCHG AX,*rw*	3	Exchange word register with AX
90+*rw*	XCHG *rw*,AX	3	Exchange with word register

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

The two operands are exchanged. The order of the operands is immaterial. BUS LOCK is asserted for the duration of the exchange, regardless of the presence or absence of the LOCK prefix or IOPL.

PROTECTED MODE EXCEPTIONS

#GP(0) if either operand is in a non-writable segment. #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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

XLAT—Table Look-up Translation

Opcode	Instruction	Clocks	Description
D7	XLAT *mb*	5	Set AL to memory byte DS: [BX + unsigned AL]
D7	XLATB	5	Set AL to memory byte DS: [BX + unsigned AL]

FLAGS MODIFIED

None

FLAGS UNDEFINED

None

OPERATION

When XLAT is executed, AL should be the unsigned index into a table addressed by DS:BX. XLAT changes the AL register from the table index into the table entry. BX is unchanged.

PROTECTED MODE EXCEPTIONS

#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.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

XOR—Logical Exclusive OR

Opcode	Instruction	Clocks	Description
30 /r	XOR *eb,rb*	2,*mem*=7	Exclusive-OR byte register into EA byte
31 /r	XOR *ew,rw*	2,*mem*=7	Exclusive-OR word register into EA word
32 /r	XOR *rb,eb*	2,*mem*=7	Exclusive-OR EA byte into byte register
33 /r	XOR *rw,ew*	2,*mem*=7	Exclusive-OR EA word into word register
34 *db*	XOR AL,*db*	3	Exclusive-OR immediate byte into AL
35 *dw*	XOR AX,*dw*	3	Exclusive-OR immediate word into AX
80 /6 *db*	XOR *eb,db*	3,*mem*=7	Exclusive-OR immediate byte into EA byte
81 /6 *dw*	XOR *ew,dw*	3,*mem*=7	Exclusive-OR immediate word into EA word

FLAGS MODIFIED

Overflow = 0, sign, zero, parity, carry = 0

FLAGS UNDEFINED

Auxiliary carry

OPERATION

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.

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, or ES segments; #SS(0) for an illegal address in the SS segment.

REAL ADDRESS MODE EXCEPTIONS

Interrupt 13 for a word operand at offset 0FFFFH.

APPENDIX C

8086/8088 COMPATIBILITY CONSIDERATIONS

SOFTWARE COMPATIBILITY CONSIDERATIONS

In general, the real address mode 80C286 will correctly execute ROM-based 8086/8088 Appensoftware. The following is a list of the minor differences between 8086 and 80C286 (Real mode).

1. Add Six Interrupt Vectors.

   The 80C286 adds six interrupts which arise only if the 8086 program has a hidden bug. These interrupts occur only for instructions which were undefined on the 8086/8088 or if a segment wraparound is attempted. It is recommended that you add an interrupt handler to the 8086 software that is to be run on the 80C286, which will treat these interrupts as invalid operations.

   This additional software does not significantly effect the existing 8086 software because the interrupts do not normally occur and should not already have been used since they are in the reserved interrupt group. Table C-1 describes the new 80C286 interrupts.

2. Do not Rely on 8086/8088 Instruction Clock Counts.

   The 80C286 takes fewer clocks for most instructions than the 8086/8088. The areas to look into are delays between I/O operations, and assumed delays in 8086/8088 operating in parallel with an 8087.

Table C-1 New 80C286 Interrupts

Interrupt Number	Function
5	A BOUND instruction was executed with a register value outside the two limit values.
6	An undefined opcode was encountered.
7	The EM bit has been set and an ESC instruction was executed. This interrupt will also occur on WAIT instructions if TS is set.
8	The interrupt table limit was changed by the LIDT instruction to a value between 20H and 43H. The default limit after reset is 3FFH, enough for all 256 interrupts.
9	A processor extension data transfer exceeded offset OFFFFH in a segment. This interrupt handler *must* execute FNINIT before *any* ESC or WAIT instruction is executed.
13	Segment wraparound was attempted by a word operation at offset OFFFFH.
16	When 80C286 attempted to execute a coprocessor instruction $\overline{\text{ERROR}}$ pin indicated an unmasked exception from previous coprocessor instruction.

3. Divide Exceptions Point at the DIV Instruction.

   Any interrupt on the 80C286 will always leave the saved CS:IP value pointing at the beginning of the instruction that failed (including prefixes). On the 8086, the CS:IP value saved for a divide exception points at the next instruction.

4. Use Interrupt 16 for Numeric Exceptions.

   Any AMD 80C287 system *must* use interrupt vector 16 for the numeric error interrupt. If an 8086/8087 or 8088/8087 system uses another vector for the 8087 interrupt, both vectors should point at the numeric error interrupt handler.

5. Numeric Exception Handlers Should allow Prefixes.

   The saved CS:IP value in the AMD 80C287 environment save area will point at any leading prefixes before an ESC instruction. On 8086/8088 systems, this value points only at the ESC instruction.

6. Do Not Attempt Undefined 8086/8088 Operations.

   Instructions like POP CS or MOV CS,op will either cause Exception 6 (undefined opcode) or perform a protection setup operation like LIDT on the 80C286. Undefined bit encodings for bits 5–3 of the second byte of POP MEM or PUSH MEM will cause Exception 13 on the 80C286.

7. Place a Far JMP Instruction at FFFF0H.

   After reset, CS:IP = F000:FFF0 on the 80C286 (versus FFFF:0000 on the 8086/8088). This change was made to allow sufficient code space to enter Protected mode without reloading CS. Placing a far JMP instruction at FFFF0H will avoid this difference. Note that the BOOTSTRAP option of LOC86 will automatically generate this jump instruction.

8. Do not Rely on the Value Written by PUSH SP.

   The 80C286 will push a different value on the stack for PUSH SP than the 8086/8088. 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 80C286.

9. Do not Shift or Rotate by More than 31 Bits.

   The 80C286 masks all shift/rotate counts to the low five bits. This MOD 32 operation limits the count to a maximum of 31 bits. With this change, the longest shift/rotate instruction is 39 clocks. Without this change, the longest shift/rotate instruction would be 264 clocks, which delays interrupt response until the instruction completes execution.

10. Do not Duplicate Prefixes.

    The 80C286 sets an instruction length limit of 10 bytes. The only way to violate this limit is by duplicating a prefix two or more times before an instruction. Exception 6 occurs if the instruction length limit is violated. The 8086/8088 has no instruction length limit.

11. Do not Rely on Odd 8086/8088 LOCK Characteristics.

    The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. The 80C286 will always assert LOCK during an XCHG instruction with memory (even if the LOCK prefix was not used). LOCK should only be used with the XCHG, MOV, MOVS, INS, and OUTS instructions.

    The 80C286 LOCK signal will *not* go active during an instruction prefetch.

12. Do not Single-Step External Interrupt Handlers.

    The priority of the 80C286 single-step interrupt is different from that of the 8086/8088. This change was made to prevent an external interrupt from being single stepped if it occurs while single stepping through a program. The 80C286 single-step interrupt has higher priority than any external interrupt.

    The 80C286 will still single step through an interrupt handler invoked by INT instructions or an instruction exception.

13. Do not Rely on IDIV Exceptions for Quotients of 80H or 8000H.

    The 80C286 can generate the largest negative number as a quotient for IDIV instructions. The 8086 will instead cause Exception 0.

14. Do not Rely on NMI Interrupting NMI Handlers.

    After an NMI is recognized, the NMI input and processor extension limit error interrupt is masked until the first IRET instruction is executed.

15. The AMD 80C287 error signal does not pass through an interrupt controller (an 8087 INT signal does). Any interrupt controller-oriented instructions for the 8087 may have to be deleted.

16. If any Real mode program relies on address space wrap around (e.g., FFF0:0400 = 0000:0300), then external hardware should be used to force the upper 4 addresses to zero during Real mode.

17. Do not use I/O ports 00F8–00FFH. These are reserved for controlling AMD 80C287 math coprocessor and future processor extensions.

HARDWARE COMPATIBILITY CONSIDERATIONS

1. Address after Reset

   8086 has CS:IP = FFFF:0000 and physical address FFFF0.

   80C286 has CS:IP = F000:FFF0 and physical address FFFFF0.

   After 80C286 reset, until the first 80C286 far JMP or far CALL, the code segment base is FF0000. This means A23–A20 will be high for CS-relative bus cycles (code fetch or use of CS override prefix) after reset until the first far JMP or far CALL instruction is performed.

2. Physical Address Formation

   In Real mode or Protected mode, the 80C286 always forms a physical address by adding a 16-bit offset with a 24-bit segment base value (8086 has 20-bit base value). Therefore, if the 80C286 in Real mode has a segment base within 64K of the top of the 1 Mb address space, and the program adds an offset of FFFFH to the segment base, the physical address will be slightly above 1Mb. Thus, to fully duplicate 1 Mb wraparound that the 8086 has, it is always necessary to force

A20 low externally when the 80C286 is in Real mode, but system hardware uses all 24 address lines.

3. LOCK signal

   On the 8086, LOCK asserted means this bus cycle is within a group of two or more locked bus cycles. On the 80C286, the LOCK signal means lock this bus cycle to the NEXT bus cycle. Therefore, on the 80C286, the LOCK signal is not asserted on the last locked bus cycle of the group of locked bus cycles.

4. Coprocessor Interface

   8086, synchronous to 8086, can become a bus master.

   AMD 80C287 math coprocessor, asynchronous to 80C286 and AMD 80C287 math coprocessor, cannot become a bus master.

   8087 pulls opcode and pointer information directly from data bus.

   80C286 passes opcode and pointer information to the AMD 80C287 math coprocessor.

   8087 uses interrupt path to signal errors to 8086.

   The AMD 80C287 math coprocessor uses dedicated ERROR signal.

   8086 requires explicit WAIT opcode preceding all ESC instructions to synchronize with 8087.

   80C286 has automatic instruction synchronization with AMD 80C287 math coprocessor.

5. Bus Cycles

   8086 has four-clock minimum bus cycle, with a time-multiplexed address/data bus.

   80C286 has two-clock minimum bus cycle, with separate buses for address and data.

APPENDIX D

MACHINE INSTRUCTION ENCODING AND DECODING

Machine instructions for the AMD 80C287 math coprocessor come in one of five different forms as shown in Table D-1. In all cases, the instructions are at least two bytes long and begin with the bit pattern 11011B, which identifies the ESCAPE class of instructions. Instructions that reference memory operands are encoded much like similar CPU instructions, because all of the CPU memory-addressing modes may be used with ESCAPE instructions.

Table D-1 AMD 80C287 Instruction Encoding

	Lower-Addressed Byte								Higher-Addressed Byte								0, 1, or 2 bytes
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
(1)	1	1	0	1	1	OP-A		1	MOD		1	OP-B		R/M			DISPLACEMENT
(2)	1	1	0	1	1	FORMAT			OP-A		MOD	OP-B		R/M			DISPLACEMENT
(3)	1	1	0	1	1	R	P		OP-A	1		1	OP-B		REG		
(4)	1	1	0	1	1	0	0	1	1	1	1	OP					
(5)	1	1	0	1	1	0	1	1	1	1	1	OP					

(1) Memory transfers, including applicable processor control instruction; 0, 1, or 2 displacement bytes may follow.
(2) Memory arithmetic and comparison instructions; 0, 1, or 2 displacement bytes may follow.
(3) Stack arithmetic and comparison instructions.
(4) Constant, transcendental, some arithmetic instructions.
(5) Processor control instructions that do not reference memory.
OP, OP-A, OP-B: Instruction opcode, possibly split into two fields.
MOD: Same as 80C286 CPU mode field.
R/M: Same as 80C286 CPU register/memory field.
FORMAT: Defines memory operand
- 00 = short real
- 01 = short integer
- 10 = long real
- 11 = word integer

R: 0 = return result to stack top
- 1 = return result to other register

P: 0 = do not pop stack
- 1 = pop stack after operation

REG: register stack element
- 000 = stack top
- 001 = next on stack
- 010 = third stack element, etc.

Note that several of the processor control instructions (see Chapter 13) may be preceded by an assembler-generated CPU WAIT instruction (encoding: 10011011B) if they are programmed using the WAIT form of their mnemonics. The assembler inserts a WAIT instruction only before these specific processor-control instructions—all of the numeric instructions are automatically synchronized by the 80C286 CPU and an explicit WAIT instruction, though allowed, is not necessary.

Table D-2 lists all AMD 80C287 machine instructions in binary sequence. This table may be used to disassemble instructions in unformatted memory dumps or instructions monitored from the data bus. Users writing exception handlers may also find this information useful to identify the offending instruction.

Table D-2 Machine Instruction Decoding Guide

1st Byte		2nd Byte	Bytes 3, 4	ASM286 Instruction Format	
Hex	Binary				
D8	1101 1000	MOD00 0R/M	(disp-lo),(disp-hi)	FADD	short-real
D8	1101 1000	MOD00 1R/M	(disp-lo),(disp-hi)	FMUL	short-real
D8	1101 1000	MOD01 0R/M	(disp-lo),(disp-hi)	FCOM	short-real
D8	1101 1000	MOD01 1R/M	(disp-lo),(disp-hi)	FCOMP	short-real
D8	1101 1000	MOD10 0R/M	(disp-lo),(disp-hi)	FSUB	short-real
D8	1101 1000	MOD10 1R/M	(disp-lo),(disp-hi)	FSUBR	short-real
D8	1101 1000	MOD11 0R/M	(disp-lo),(disp-hi)	FDIV	short-real
D8	1101 1000	MOD11 1R/M	(disp-lo),(disp-hi)	FDIVR	short-real
D8	1101 1000	1100 0REG		FADD	ST,ST(i)
D8	1101 1000	1100 1REG		FMUL	ST,ST(i)
D8	1101 1000	1101 0REG		FCOM	ST(i)
D8	1101 1000	1101 1REG		FCOMP	ST(i)
D8	1101 1000	1110 0REG		FSUB	ST,ST(i)
D8	1101 1000	1110 1REG		FSUBR	ST,ST(i)
D8	1101 1000	1111 0REG		FDIV	ST,ST(i)
D8	1101 1000	1111 1REG		FDIVR	ST,ST(i)
D9	1101 1001	MOD00 0R/M	(disp-lo),(disp-hi)	FLD	short-real
D9	1101 1001	MOD00 1R/M		reserved	
D9	1101 1001	MOD01 0R/M	(disp-lo),(disp-hi)	FST	short-real
D9	1101 1001	MOD01 1R/M	(disp-lo),(disp-hi)	FSTP	short-real
D9	1101 1001	MOD10 0R/M	(disp-lo),(disp-hi)	FLDENV	14-bytes
D9	1101 1001	MOD10 1R/M	(disp-lo),(disp-hi)	FLDCW	2-bytes
D9	1101 1001	MOD11 0R/M	(disp-lo),(disp-hi)	FSTENV	14-bytes
D9	1101 1001	MOD11 1R/M	(disp-lo),(disp-hi)	FSTCW	2-bytes
D9	1101 1001	1100 0REG		FLD	ST(i)
D9	1101 1001	1100 1REG		FXCH	ST(i)
D9	1101 1001	1101 0000		FNOP	
D9	1101 1001	1101 0001		reserved	
D9	1101 1001	1101 001-		reserved	
D9	1101 1001	1101 01--		reserved	
D9	1101 1001	1101 1REG		*(1)	
D9	1101 1001	1110 0000		FCHS	
D9	1101 1001	1110 0001		FABS	
D9	1101 1001	1110 001-		reserved	
D9	1101 1001	1110 0100		FTST	
D9	1101 1001	1110 0101		FXAM	
D9	1101 1001	1110 011-		reserved	
D9	1101 1001	1110 1000		FLD1	

Table D-2 Machine Instruction Decoding Guide (continued)

1st Byte		2nd Byte	Bytes 3, 4	ASM286 Instruction Format	
Hex	Binary				
D9	1101 1001	1110 1001		FLDL2T	
D9	1101 1001	1110 1010		FLDL2E	
D9	1101 1001	1110 1011		FLDPI	
D9	1101 1001	1110 1100		FLDLG2	
D9	1101 1001	1110 1101		FLDLN2	
D9	1101 1001	1110 1110		FLDZ	
D9	1101 1001	1110 1111		reserved	
D9	1101 1001	1111 0000		F2XM1	
D9	1101 1001	1111 0001		FYL2X	
D9	1101 1001	1111 0010		FPTAN	
D9	1101 1001	1111 0011		FPATAN	
D9	1101 1001	1111 0100		FXTRACT	
D9	1101 1001	1111 0101		reserved	
D9	1101 1001	1111 0110		FDECSTP	
D9	1101 1001	1111 0111		FINCSTP	
D9	1101 1001	1111 1000		FPREM	
D9	1101 1001	1111 1001		FYL2XP1	
D9	1101 1001	1111 1010		FSQRT	
D9	1101 1001	1111 1011		reserved	
D9	1101 1001	1111 1100		FRNDINT	
D9	1101 1001	1111 1101		FSCALE	
D9	1101 1001	1111 111-		reserved	
DA	1101 1010	MOD00 0R/M	(disp-lo),(disp-hi)	FIADD	short-integer
DA	1101 1010	MOD00 1R/M	(disp-lo),(disp-hi)	FIMUL	short-integer
DA	1101 1010	MOD01 0R/M	(disp-lo),(disp-hi)	FICOM	short-integer
DA	1101 1010	MOD01 1R/M	(disp-lo),(disp-hi)	FICOMP	short-integer
DA	1101 1010	MOD10 0R/M	(disp-lo),(disp-hi)	FISUB	short-integer
DA	1101 1010	MOD10 1R/M	(disp-lo),(disp-hi)	FISUBR	short-integer
DA	1101 1010	MOD11 0R/M	(disp-lo),(disp-hi)	FIDIV	short-integer
DA	1101 1010	MOD11 1R/M	(disp-lo),(disp-hi)	FIDIVR	short-integer
DA	1101 1010	11-- ----		reserved	
DB	1101 1011	MOD00 0R/M	(disp-lo),(disp-hi)	FILD	short-integer
DB	1101 1011	MOD00 1R/M	(disp-lo),(disp-hi)	reserved	
DB	1101 1011	MOD01 0R/M	(disp-lo),(disp-hi)	FIST	short-integer
DB	1101 1011	MOD01 1R/M	(disp-lo),(disp-hi)	FISTP	short-integer
DB	1101 1011	MOD10 0R/M	(disp-lo),(disp-hi)	reserved	
DB	1101 1011	MOD10 1R/M	(disp-lo),(disp-hi)	FLD	temp-real
DB	1101 1011	MOD11 0R/M	(disp-lo),(disp-hi)	reserved	
DB	1101 1011	MOD11 1R/M	(disp-lo),(disp-hi)	FSTP	temp-real
DB	1101 1011	110- ----		reserved	
DB	1101 1011	1110 0000		reserved (8087 FENI)	
DB	1101 1011	1110 0001		reserved (8087 FDISI)	
DB	1101 1011	1110 0010		FCLEX	
DB	1101 1011	1110 0011		FINIT	
DB	1101 1011	1110 0100		FSETPM	
DB	1101 1011	1110 1---		reserved	
DB	1101 1011	1111 ----		reserved	
DC	1101 1100	MOD00 0R/M	(disp-lo),(disp-hi)	FADD	long-real
DC	1101 1100	MOD00 1R/M	(disp-lo),(disp-hi)	FMUL	long-real
DC	1101 1100	MOD01 0R/M	(disp-lo),(disp-hi)	FCOM	long-real
DC	1101 1100	MOD01 1R/M	(disp-lo),(disp-hi)	FCOMP	long-real
DC	1101 1100	MOD10 0R/M	(disp-lo),(disp-hi)	FSUB	long-real

Table D–2 Machine Instruction Decoding Guide (continued)

1st Byte		2nd Byte	Bytes 3, 4	ASM286 Instruction Format	
Hex	Binary				
DC	1101 1100	MOD10 1R/M	(disp-lo),(disp-hi)	FSUBR	long-real
DC	1101 1100	MOD11 0R/M	(disp-lo),(disp-hi)	FDIV	long-real
DC	1101 1100	MOD11 1R/M	(disp-lo),(disp-hi)	FDIVR	long-real
DC	1101 1100	1100 0REG		FADD	ST(i),ST
DC	1101 1100	1100 1REG		FMUL	ST(i),ST
DC	1101 1100	1101 0REG		*(2)	
DC	1101 1100	1101 1REG		*(3)	
DC	1101 1100	1110 0REG		FSUB	ST(i),ST
DC	1101 1100	1110 1REG		FSUBR	ST(i),ST
DC	1101 1100	1111 0REG		FDIV	ST(i),ST
DC	1101 1100	1111 1REG		FDIVR	ST(i),ST
DD	1101 1101	MOD00 0R/M	(disp-lo),(disp-hi)	FLD	long-real
DD	1101 1101	MOD00 1R/M		reserved	
DD	1101 1101	MOD01 0R/M	(disp-lo),(disp-hi)	FST	long-real
DD	1101 1101	MOD01 1R/M	(disp-lo),(disp-hi)	FSTP	long-real
DD	1101 1101	MOD10 0R/M	(disp-lo),(disp-hi)	FRSTOR	94-bytes
DD	1101 1101	MOD10 1R/M	(disp-lo),(disp-hi)	reserved	
DD	1101 1101	MOD11 0R/M	(disp-lo),(disp-hi)	FSAVE	94-bytes
DD	1101 1101	MOD11 1R/M	(disp-lo),(disp-hi)	FSTSW	2-bytes
DD	1101 1101	1100 0REG		FFREE	ST(i)
DD	1101 1101	1100 1REG		*(4)	
DD	1101 1101	1101 0REG		FST	ST(i)
DD	1101 1101	1101 1REG		FSTP	ST(i)
DD	1101 1101	111- ----		reserved	
DE	1101 1110	MOD00 0R/M	(disp-lo),(disp-hi)	FIADD	word-integer
DE	1101 1110	MOD00 1R/M	(disp-lo),(disp-hi)	FIMUL	word-integer
DE	1101 1110	MOD01 0R/M	(disp-lo),(disp-hi)	FICOM	word-integer
DE	1101 1110	MOD01 1R/M	(disp-lo),(disp-hi)	FICOMP	word-integer
DE	1101 1110	MOD10 0R/M	(disp-lo),(disp-hi)	FISUB	word-integer
DE	1101 1110	MOD10 1R/M	(disp-lo),(disp-hi)	FISUBR	word-integer
DE	1101 1110	MOD11 0R/M	(disp-lo),(disp-hi)	FIDIV	word-integer
DE	1101 1110	MOD11 1R/M	(disp-lo),(disp-hi)	FIDIVR	word-integer
DE	1101 1110	1100 0REG		FADDP	ST(i),ST
DE	1101 1110	1100 1REG		FMULP	ST(i),ST
DE	1101 1110	1101 0---		*(5)	
DE	1101 1110	1101 1000		reserved	
DE	1101 1110	1101 1001		FCOMPP	
DE	1101 1110	1101 101-		reserved	
DE	1101 1110	1101 11--		reserved	
DE	1101 1110	1110 0REG		FSUBP	ST(i),ST
DE	1101 1110	1110 1REG		FSUBRP	ST(i),ST
DE	1101 1110	1111 0REG		FDIVP	ST(i),ST
DE	1101 1110	1111 1REG		FDIVRP	ST(i),ST
DF	1101 1111	MOD00 0R/M	(disp-lo),(disp-hi)	FILD	word-integer
DF	1101 1111	MOD00 1R/M	(disp-lo),(disp-hi)	reserved	
DF	1101 1111	MOD01 0R/M	(disp-lo),(disp-hi)	FIST	word-integer
DF	1101 1111	MOD01 1R/M	(disp-lo),(disp-hi)	FISTP	word-integer
DF	1101 1111	MOD10 0R/M	(disp-lo),(disp-hi)	FBLD	packed-decimal
DF	1101 1111	MOD10 1R/M	(disp-lo),(disp-hi)	FILD	long-integer
DF	1101 1111	MOD11 0R/M	(disp-lo),(disp-hi)	FBSTP	packed-decimal
DF	1101 1111	MOD11 1R/M	(disp-lo),(disp-hi)	FISTP	long-integer
DF	1101 1111	1100 0REG		*(6)	

Table D-2 Machine Instruction Decoding Guide (continued)

1st Byte		2nd Byte	Bytes 3, 4	ASM286 Instruction Format
Hex	Binary			
DF	1101 1111	1100 1REG		*(7)
DF	1101 1111	1101 0REG		*(8)
DF	1101 1111	1101 1REG		*(9)
DF	1101 1111	1110 000		FSTSW AX
DF	1101 1111	1111 XXX		reserved

NOTE:

* The marked encodings are *not* generated by the language translators. If, however, the 80287 encounters one of these encodings in the instruction stream, it will execute it as follows:

(1) FSTP ST(i)

(2) FCOM ST(i)

(3) FCOMP ST(i)

(4) FXCH ST(i)

(5) FCOMP ST(i)

(6) FFREE ST(i) and pop stack

(7) FXCH ST(i)

(8) FSTP ST(i)

(9) FSTP ST(i)

APPENDIX E

COMPATIBILITY BETWEEN THE AMD 80C287 MATH COPROCESSOR AND THE 8087

The 80C286/AMD 80C287 math coprocessor operating in Real-Address mode will execute 8087 programs without major modification. However, because of differences in the handling of numeric exceptions by the AMD 80C287 and the 8087, exception-handling routines may need to be changed.

This appendix summarizes the differences between the AMD 80C287 math coprocessor and the 8087, and provides details showing how 8087 programs can be ported to the AMD 80C287 math coprocessor.

- The AMD 80C287 math coprocessor signals exceptions through a dedicated $\overline{\text{ERROR}}$ line to the 80C286. The AMD 80C287 error signal does not pass through an interrupt controller (the 8087 INT signal does). Therefore, any interrupt-controller-oriented instructions in numeric exception handlers for the 8087 should be deleted.
- The 8087 instructions FENI/FNENI and FDISI/FNDISI perform no useful function in the AMD 80C287 math coprocessor. If the AMD 80C287 math coprocessor encounters one of these opcodes in its instruction stream, the instruction will effectively be ignored—none of the AMD 80C287 internal states will be updated. While 8087 code containing these instructions may be executed on the AMD 80C287 math coprocessor, it is unlikely that the exception-handling routines containing these instructions will be completely portable to the AMD 80C287 math coprocessor.
- Interrupt vector 16 must point to the numeric exception handling routine.
- The ESC instruction address saved in the AMD 80C287 math coprocessor includes any leading prefixes before the ESC opcode. The corresponding address saved in the 8087 does not include leading prefixes.
- In Protected-Address mode, the format of the AMD 80C287 math coprocessor's saved instruction and address pointers is different than for the 8087. The instruction opcode is not saved in Protected mode—exception handlers will have to retrieve the opcode from memory if needed.
- Interrupt 7 will occur in the 80C286 when executing ESC instructions with either TS (task switched) or EM (emulation) of the 80C286 MSW set (TS = 1 or EM = 1). If TS is set, then a WAIT instruction will also cause Interrupt 7. An exception handler should be included in AMD 80C287 code to handle these situations.
- Interrupt 9 will occur if the second or subsequent words of a floating-point operand fall outside a segment's size. Interrupt 13 will occur if the starting address of a numeric operand falls outside a segment's size. An exception handler should be included in AMD 80C287 code to report these programming errors.
- Except for the processor control instructions, all of the AMD 80C287 numeric instructions are automatically synchronized by the 80C286 CPU—the 80C286 automatically tests the $\overline{\text{BUSY}}$ line from the AMD 80C287 math coprocessor to ensure that the AMD 80C287 math coprocessor has completed its previous instruction before executing the next ESC instruction. No explicit WAIT instruc-

tions are required to assure this synchronization. For the 8087 used with 8086 and 8088 processors, explicit WAITs are required before each numeric instruction to ensure synchronization. Although 8087 programs having explicit WAIT instructions will execute perfectly on the AMD 80C287 math coprocessor without reassembly, these WAIT instructions are unnecessary.

- Since the AMD 80C287 math coprocessor does not require WAIT instructions before each numeric instruction, the ASM286 assembler does not automatically generate these WAIT instructions. The ASM86 assembler, however, automatically precedes every ESC instruction with a WAIT instruction. Although numeric routines generated using the ASM86 assembler will generally execute correctly on the 80C286/20, reassembly using ASM286 may result in a more compact code image.

  The processor control instructions for the AMD 80C287 math coprocessor may be coded using either a WAIT or No-WAIT form of mnemonic. The WAIT forms of these instructions cause ASM286 to precede the ESC instruction with a CPU WAIT instruction, in the identical manner as does ASM86.

- A recommended way to detect the presence of an AMD 80C287 math coprocessor in an 80C286 system is shown below. It assumes that the system hardware causes the data bus to be high if no AMD 80C287 math coprocessor is present to drive the data lines during the FSTSW (Store AMD 80C287 Status Word) instruction.

```
FND_287: FNINIT        ; initialize numeric processor.
     FSTSTW STAT       ; store status word into location
                       ; STAT.
     MOV   AX,STAT
     OR    AL,AL       ; Zero Flag reflects result of OR.
     JZ    GOT_287     ; Zero in AL means AMD 80C287 is present.
;
;  No AMD 80C287 Present
;
     SMSW  AX
     OR    AX,0004H    ; set EM bit in Machine Status Word.
     LMSW  AX          ; to enable software emulation of 287.
     JMP   CONTINUE
;
;  AMD 80C287 math coprocessor is present in system
;
GOT_287: SMSW  AX
     OR    AX,0002H    ; set MP bit in Machine Status Word
     LMSW  AX          ; to permit normal AMD 80C287 operation
;
;  Continue . . .
;
CONTINUE:              ; and off we go
```

A design for the 80C286 with the AMD 80C287 extension must place a pullup resistor on one of the low eight data bus bits of the 80C286 to be sure it is read as a high when no AMD 80C287 math coprocessor is present.

APPENDIX F

IMPLEMENTING THE IEEE P754 STANDARD

This appendix describes the relationship between the AMD 80C287 math coprocessor and the IEEE Standard. Where the Standard has options, the choices in implementing the AMD 80C287 math coprocessor are described. Where portions of the Standard are implemented through software, this appendix indicates which modules of the AMD 80C287 Support Library implement the Standard. Where special software in addition to the Support Library may be required by your application, this appendix indicates how to write this software.

This appendix contains many terms with precise technical meanings, specified in the 754 Standard. Where these terms are used, they have been capitalized to emphasize the precision of their meanings. The Glossary provides the definitions for all capitalized phrases in this appendix.

OPTIONS IMPLEMENTED IN THE AMD 80C287 MATH COPROCESSOR

The AMD 80C287 SHORT_REAL and LONG_REAL formats conform precisely to the Standard's Single and Double Floating-Point Numbers, respectively. The AMD 80C287 TEMP_REAL format is the same as the Standard's Double Extended format. The Standard allows a choice of Bias in representing the exponent; the AMD 80C287 math coprocessor uses the Bias 16383 decimal.

For the Double Extended format, the Standard contains an option for the meaning of the minimum exponent combined with a nonzero significand. The Bias for this special case can be either 16383, as in all the other cases, or 16382, making the smallest exponent equivalent to the second-smallest exponent. The AMD 80C287 math coprocessor uses the Bias 16382 for this case. This allows the AMD 80C287 math coprocessor to distinguish between Denormal numbers (integer part is zero, fraction is nonzero, Biased exponent is 0) and Unnormal numbers of the same value (same as the denormal except the Biased Exponent is 1).

The Standard allows flexibility in specifying which NaNs are trapping and which are nontrapping. The EH287.LIB module of the AMD 80C287 Support Library provides a software implementation of nontrapping NaNs, and defines one distinction between trapping and nontrapping NaNs: If the most significant bit of the fractional part of a NaN is 1, the NaN is nontrapping. If it is 0, the NaN is trapping.

When a masked Invalid Operation error involves two NaN inputs, the Standard allows flexibility in choosing which NaN is output. The AMD 80C287 math coprocessor selects the NaN whose absolute value is greatest.

AREAS OF THE STANDARD IMPLEMENTED IN SOFTWARE

There are five areas of the Standard that are not implemented directly in the AMD 80C287 hardware; these areas are instead implemented in software as part of the AMD 80C287 Support Library.

1. The Standard requires that a Normalizing Mode be provided, in which any non-normal operands to functions are automatically normalized before the function is performed. The AMD 80C287 math coprocessor provides a Denormal Operand exception for this case, allowing the exception handler the opportunity to perform the normalization specified by the Standard. The Denormal operand exception handler provided by EH287.LIB implements the Standard's Normalizing Mode completely for Single- and Double-precision arguments. Normalizing mode for Double Extended operands is implemented in EH287.LIB with one non-Standard feature, discussed in the next section.
2. The Standard specifies that in comparing two operands whose relationship is unordered, the equality test yield an answer of FALSE, with no errors or exceptions. The AMD 80C287 FCOM and FTST instructions themselves issue an Invalid Operation exception in this case. The error handler EH287.LIB filters out this Invalid Operation error using the following convention: Whenever an FCOM or FTST instruction is followed by a MOV AX,AX instruction (8BC0 Hex), and neither argument is a trapping NaN, the error handler will assume that a Standard equality comparison was intended, and return the correct answer with the Invalid Operation exception flag erased. Note that the Invalid Operation exception must be unmasked for this action to occur.
3. The Standard requires that two kinds of NaN's be provided: trapping and non-trapping. Nontrapping NaNs will not cause further Invalid Operation errors when they occur as operands to calculations. The AMD 80C287 hardware directly supports only trapping NaN's; the EH287.LIB software implements nontrapping NaNs by returning the correct answer with the Invalid Operation exception flag erased. Note that the Invalid Operation exception must be unmasked for this action to occur.
4. The Standard requires that all functions that convert real numbers to integer formats automatically normalize the inputs if necessary. The integer conversion functions contained in CEL287.LIB fully meet the Standard in this respect; the AMD 80C287 FIST instruction alone does not perform this normalization.
5. The Standard specifies the remainder function which is provided by mqerRMD in CEL287.LIB. The AMD 80C287 FPREM instruction returns answers within a different range.

ADDITIONAL SOFTWARE TO MEET THE STANDARD

There are two cases in which additional software is required in conjunction with the AMD 80C287 Support Library in order to meet the standard. The AMD 80C287 Support Library does not provide this software in the interest of saving space and because the vast majority of applications will never encounter these cases.

1. When the Invalid Operation exception is masked, Nontrapping NaNs are not implemented fully. Likewise, the Standard's equality test for unordered operands is not implemented when the Invalid Operation exception is masked. Programmers can simulate the Standard notion of a masked Invalid Operation exception by unmasking the AMD 80C287 Invalid Operation exception, and providing an Invalid Operation exception handler that supports nontrapping NaNs and the equality test, but otherwise acts just as if the Invalid Operation exception were masked.
2. In Normalizing Mode, Denormal operands in the TEMP_REAL format are converted to 0 by EH287.LIB, giving sharp Underflow to 0. The Standard specifies that the operation be performed on the real numbers represented by the denor-

mals, giving gradual underflow. To correctly perform such arithmetic while in Normalizing Mode, programmers would have to normalize the operands into a format identical to TEMP_REAL except for two extra exponent bits, then perform the operation on those numbers. Thus, software must be written to handle the 17-bit exponent explicitly.

In designing the EH287.LIB, it was felt that it would be a disadvantage to most users to increase the size of the Normalizing routine by the amount necessary to provide this expanded arithmetic. Because the TEMP_REAL exponent field is so much larger than the LONG_REAL exponent field, it is extremely unlikely that TEMP_REAL underflow will be encountered in most applications.

If meeting the standard is a more important criterion for your application than the choice between Normalizing and warning modes, then you can select warning mode (Denormal Operand exceptions masked), which fully meets the Standard.

If you do wish to implement the Normalization of denormal operands in TEMP_REAL format using extra exponent bits, the list below indicates some useful pointers about handling Denormal Operand exceptions:

- TEMP_REAL numbers are considered Denormal by the AMD 80C287 math coprocessor whenever the Biased Exponent is 0 (minimum exponent). This is true even if the explicit integer bit of the significand is 1. Such numbers can occur as the result of Underflow.
- The AMD 80C287 FLD instruction can cause a Denormal Operand error if a number is being loaded from memory. It will not cause this exception if the number is being loaded from elsewhere in the AMD 80C287 stack.
- The AMD 80C287 FCOM and FTST instructions will cause a Denormal Operand exception for unnormal operands as well as for denormal operands.
- In cases where both the Denormal Operand and Invalid Operation exceptions occur, you will want to know which is signaled first. When a comparison instruction operates between a nonexistent stack element and a denormal number in 80C286 memory, the D and I exceptions are issued simultaneously. In all other situations, a Denormal Operand exception takes precedence over a nonstack Invalid Operation exception, while a stack Invalid Operation exception takes precedence over a Denormal Operand exception.

AMD 80C287 MATH COPROCESSOR AND FLOATING-POINT TERMINOLOGY

This glossary defines many terms that have precise technical meanings as specified in the IEEE 754 Standard. Where these terms are used, they have been capitalized to emphasize the precision of their meanings.

Affine Mode: a state of the AMD 80C287 math coprocessor, selected in the AMD 80C287 Control Word, in which infinities are treated as having a sign. Thus, the values +INFINITY and –INFINITY are considered different; they can be compared with finite numbers and with each other.

Base: (1) a term used in logarithms and exponentials. In both contexts, it is a number that is being raised to a power. The two equations (y = log base b of x) and (b^y = x) are the same.

Base: (2) a number that defines the representation being used for a string of digits. Base 2 is the binary representation; Base 10 is the decimal representation; Base 16 is the hexadecimal representation. In each case, the Base is the factor of increased significance for each succeeding digit (working up from the bottom).

Bias: the difference between the unsigned Integer that appears in the Exponent field of a Floating-Point Number and the true Exponent that it represents. To obtain the true Exponent, you must subtract the Bias from the given Exponent. For example, the Short Real format has a Bias of 127 whenever the given Exponent is nonzero. If the 8-bit Exponent field contains 10000011, which is 131, the true Exponent is 131–127, or +4.

Biased Exponent: the Exponent as it appears in a Floating-Point Number, interpreted as an unsigned, positive number. In the above example, 131 is the Biased Exponent.

Binary Coded Decimal: a method of storing numbers that retains a base 10 representation. Each decimal digit occupies four full bits (one hexadecimal digit). The hex values A through F (1010 through 1111) are not used. The AMD 80C287 math coprocessor supports a Packed Decimal format that consists of nine bytes of Binary Coded Decimal (18 decimal digits) and one sign byte.

Binary Point: an entity just like a decimal point, except that it exists in binary numbers. Each binary digit to the right of the Binary Point is multiplied by an increasing negative power of two.

C3–C0: the four "condition code" bits of the AMD 80C287 Status Word. These bits are set to certain values by the compare, test, examine, and remainder functions of the AMD 80C287 math coprocessor.

Characteristic: a term used for some computers, meaning the Exponent field of a Floating-Point Number.

Chop: to set the fractional part of a real number to zero, yielding the nearest integer in the direction of zero.

Control Word: a 16-bit AMD 80C287 register that the user can set, to determine the modes of computation the AMD 80C287 math coprocessor will use, and the error interrupts that will be enabled.

Denormal: a special form of Floating-Point Number, produced when an Underflow occurs. On the AMD 80C287 math coprocessor, a Denormal is defined as a number with a Biased Exponent that is zero. By providing a Significand with leading zeros, the range of possible negative Exponents can be extended by the number of bits in the Significand. Each leading zero is a bit of lost accuracy, so the extended Exponent range is obtained by reducing significance.

Double Extended: the Standard's term for the AMD 80C287 Temporary Real format, with more Exponent and Significand bits than the Double (Long Real) format, and an explicit Integer bit in the Significand.

Double Floating Point Number: the Standard's term for the AMD 80C287 math coprocessor's 64-bit Long Real format.

Environment: the 14 bytes of AMD 80C287 registers affected by the FSTENV and FLDENV instructions. It encompasses the entire state of the AMD 80C287 math coprocessor, except for the 8 Temporary Real numbers of the AMD 80C287 stack. Included are the Control Word, Status Word, Tag Word, and the instruction, opcode, and operand information provided by interrupts.

Exception: any of the six error conditions (I, D, O, U, Z, P) signaled by the AMD 80C287 math coprocessor.

Exponent: (1) any power that is raised by an exponential function. For example, the operand to the function mqerEXP is an Exponent. The Integer operand to mqerYI2 is an Exponent.

Exponent: (2) the field of a Floating-Point Number that indicates the magnitude of the number. This would fall under the above more general definition (1), except that a Bias sometimes needs to be subtracted to obtain the correct power.

Floating-Point Number: a sequence of data bytes that, when interpreted in a standardized way, represents a Real number. Floating-Point Numbers are more versatile than Integer representations in two ways. First, they include fractions. Second, their Exponent parts allow a much wider range of magnitude than possible with fixed-length Integer representations.

Gradual Underflow: a method of handling the Underflow error condition that minimizes the loss of accuracy in the result. If there is a Denormal number that represents the correct result, that Denormal is returned. Thus, digits are lost only to the extent of denormalization. Most computers return zero when Underflow occurs, losing all significant digits.

Implicit Integer Bit: a part of the Significand in the Short Real and Long Real formats that is not explicitly given. In these formats, the entire given Significand is considered to be to the right of the Binary Point. A single Implicit Integer Bit to the left of the Binary Point is always 1, except in one case. When the Exponent is the minimum (Biased Exponent is 0), the Implicit Integer Bit is 0.

Indefinite: a special value that is returned by functions when the inputs are such that no other sensible answer is possible. For each Floating-Point format there exists one Nontrapping NaN that is designated as the Indefinite value. For binary Integer formats, the negative number furthest from zero is often considered the Indefinite value. For the AMD 80C287 Packed Decimal format, the Indefinite value contains all 1's in the sign byte and the uppermost digits byte.

Infinity: a value that has greater magnitude than any Integer or any Real number. The existence of Infinity is subject to heated philosophical debate. However, it is often useful to consider Infinity as another number, subject to special rules of arithmetic. All three Floating-Point formats provide representations for +INFINITY and –INFINITY. They support two ways of dealing with Infinity: Projective (unsigned) and Affine (signed).

Integer: a number (positive, negative, or zero) that is finite and has no fractional part. Integer can also mean the computer representation for such a number: a sequence of data bytes, interpreted in a standard way. It is perfectly reasonable for Integers to be represented in a Floating-Point format; this is what the AMD 80C287 math coprocessor does whenever an Integer is pushed onto the AMD 80C287 stack.

Invalid Operation: the error condition for the AMD 80C287 math coprocessor that covers all cases not covered by other errors. Included are AMD 80C287 stack overflow and underflow, NaN inputs, illegal infinite inputs, out-of-range inputs, and illegal unnormal inputs.

Long Integer: an Integer format supported by the AMD 80C287 math coprocessor that consists of a 64-bit Two's Complement quantity.

Long Real: a Floating-Point Format supported by the AMD 80C287 math coprocessor that consists of a sign, an 11-bit Biased Exponent, an Implicit Integer Bit, and a 52-bit Significand—a total of 64 explicit bits.

Mantissa: a term used for some computers, meaning the Significand of a Floating-Point Number.

Masked: a term that applies to each of the six AMD 80C287 Exceptions I,D,Z,O,U,P. An exception is Masked if a corresponding bit in the AMD 80C287 Control Word is set to 1. If an exception is Masked, the AMD 80C287 math coprocessor will not generate an interrupt when the error condition occurs; it will instead provide its own error recovery.

NaN: an abbreviation for Not a Number; a Floating-Point quantity that does not represent any numeric or infinite quantity. NaNs should be returned by functions that encounter serious errors. If created during a sequence of calculations, they are transmitted to the final answer and can contain information about where the error occurred.

Nontrapping NaN: a NaN in which the most significant bit of the fractional part of the Significand is 1. By convention, these NaNs can undergo certain operations without visible error. Nontrapping NaNs are implemented for the AMD 80C287 math coprocessor via the software in EH87.LIB.

Normal: the representation of a number in a Floating-Point format in which the Significand has an Integer bit 1 (either explicit or Implicit).

Normalizing Mode: a state in which nonnormal inputs are automatically converted to normal inputs whenever they are used in arithmetic. Normalizing Mode is implemented for the AMD 80C287 math coprocessor via the software in EH87.LIB.

Overflow: an error condition in which the correct answer is finite, but has magnitude too great to be represented in the destination format.

Packed Decimal: an Integer format supported by the AMD 80C287 math coprocessor. A Packed Decimal number is a 10-byte quantity, with nine bytes of 18 Binary Coded Decimal digits, and one byte for the sign.

Pop: to remove from a stack the last item that was placed on the stack.

Precision Control: an option, programmed through the AMD 80C287 Control Word, that allows all AMD 80C287 arithmetic to be performed with reduced precision. Because no speed advantage results from this option, its only use is for strict compatibility with the IEEE Standard, and with other computer systems.

Precision Exception: an AMD 80C287 error condition that results when a calculation does not return an exact answer. This exception is usually Masked and ignored; it is used only in extremely critical applications, when the user must know if the results are exact.

Projective Mode: a state of the AMD 80C287 math coprocessor, selected in the AMD 80C287 Control Word, in which infinities are treated as not having a sign. Thus the values +INFINITY and –INFINITY are considered the same. Certain operations, such as comparison to finite numbers, are illegal in Projective Mode but legal in Affine Mode. Thus Projective Mode gives you a greater degree of error control over infinite inputs.

Pseudo Zero: a special value of the Temporary Real format. It is a number with a zero significand and an Exponent that is neither all zeros or all ones. Pseudo zeros can come about as the result of multiplication of two Unnormal numbers; but they are very rare.

Real: any finite value (negative, positive, or zero) that can be represented by a decimal expansion. The fractional part of the decimal expansion can contain an infinite number of digits. Reals can be represented as the points of a line marked off like a ruler. The term Real can also refer to a Floating-Point Number that represents a Real value.

Short Integer: an Integer format supported by the AMD 80C287 math coprocessor that consists of a 32-bit Two's Complement quantity. Short Integer is not the shortest AMD 80C287 Integer format —the 16-bit Word Integer is.

Short Real: a Floating-Point Format supported by the AMD 80C287 math coprocessor, which consists of a sign, an 8-bit Biased Exponent, an Implicit Integer Bit, and a 23-bit Significand—a total of 32 explicit bits.

Significand: the part of a Floating-Point Number that consists of the most significant nonzero bits of the number, if the number were written out in an unlimited binary format. The Significand alone is considered to have a Binary Point after the first (possibly Implicit) bit; the Binary Point is then moved according to the value of the Exponent.

Single Extended: a Floating-Point format, required by the Standard, that provides greater precision than Single; it also provides an explicit Integer Significand bit. The AMD 80C287 math coprocessor's Temporary Real format meets the Single Extended requirement as well as the Double Extended requirement.

Single Floating-Point Number: the Standard's term for the AMD 80C287 math coprocessor's 32-bit Short Real format.

Standard: a Proposed Standard for Binary Floating-Point Arithmetic, Draft 10.0 of IEEE Task P754, December 2, 1982.

Status Word: A 16-bit AMD 80C287 register that can be manually set, but which is usually controlled by side effects to AMD 80C287 instructions. It contains condition codes, the AMD 80C287 stack pointer, busy and interrupt bits, and error flags.

Tag Word: a 16-bit AMD 80C287 register that is automatically maintained by the AMD 80C287 math coprocessor. For each space in the AMD 80C287 stack, it tells if the space is occupied by a number; if so, it gives information about what kind of number.

Temporary Real: the main Floating-Point Format used by the AMD 80C287 math coprocessor. It consists of a sign, a 15-bit Biased Exponent, and a Significand with an explicit Integer bit and 63 fractional-part bits.

Transcendental: one of a class of functions for which polynomial formulas are always approximate, never exact for more than isolated values. The AMD 80C287 math coprocessor supports trigonometric, exponential, and logarithmic functions; all are Transcendental.

Trapping NaN: a NaN that causes an I error whenever it enters into a calculation or comparison, even a nonordered comparison.

Two's Complement: a method of representing Integers. If the uppermost bit is 0, the number is considered positive, with the value given by the rest of the bits. If the uppermost bit is 1, the number is negative, with the value obtained by subtracting ($2^{\text{bit count}}$) from all the given bits. For example, the 8-bit number 11111100 is –4, obtained by subtracting 2^8 from 252.

Unbiased Exponent: the true value that tells how far and in which direction to move the Binary Point of the Significand of a Floating-Point Number. For example, if a Short Real Exponent is 131, we subtract the Bias 127 to obtain the Unbiased Exponent +4. Thus, the Real number being represented is the Significand with the Binary Point shifted 4 bits to the right.

Underflow: an error condition in which the correct answer is nonzero, but has a magnitude too small to be represented as a Normal number in the destination Floating-Point format. The Standard specifies that an attempt be made to represent the number as a Denormal.

Unmasked: a term that applies to each of the six AMD 80C287 Exceptions: I,D,Z,O,U,P. An exception is Unmasked if a corresponding bit in the AMD 80C287 Control Word is set to 0. If an exception is Unmasked, the AMD 80C287 math coprocessor will generate an interrupt when the error condition occurs. You can provide an interrupt routine that customizes your error recovery.

Unnormal: a Temporary Real representation in which the explicit Integer bit of the Significand is zero, and the exponent is nonzero. We consider Unnormal numbers distinct from Denormal numbers.

Word Integer: an Integer format supported by both the 80C286 and the AMD 80C287 math coprocessor that consists of a 16-bit Two's Complement quantity.

Zero divide: an error condition in which the inputs are finite, but the correct answer, even with an unlimited exponent, has infinite magnitude.

INDEX

Sales Offices

North American

ALABAMA		(205) 882-9122
ARIZONA		(602) 242-4400
CALIFORNIA,		
Culver City		(213) 645-1524
Newport Beach		(714) 752-6262
Sacramento(Roseville)		(916) 786-6700
San Diego		(619) 560-7030
San Jose		(408) 452-0500
Woodland Hills		(818) 992-4155
CANADA, Ontario,		
Kanata		(613) 592-0060
Willowdale		(416) 224-5193
COLORADO		(303) 741-2900
CONNECTICUT		(203) 264-7800
FLORIDA,		
Clearwater		(813) 530-9971
Ft. Lauderdale		(305) 776-2001
Orlando (Longwood)		(407) 862-9292
GEORGIA		(404) 449-7920
ILLINOIS,		
Chicago (Itasca)		(708) 773-4422
Naperville		(708) 505-9517
KANSAS		(913) 451-3115
MARYLAND		(301) 381-3790
MASSACHUSETTS		(617) 273-3970
MINNESOTA		(612) 938-0001
NEW JERSEY,		
Cherry Hill		(609) 662-2900
Parsippany		(201) 299-0002
NEW YORK,		
Liverpool		(315) 457-5400
Brewster		(914)279-8323
Rochester		(716) 272-9020
NORTH CAROLINA		
Harrisburg		(704) 455-1010
Raleigh		(919) 878-8111
OHIO,		
Columbus (Westerville)		(614) 891-6455
Dayton		(513) 439-0268
OREGON		(503) 245-0080
PENNSYLVANIA		(215) 398-8006
SOUTH CAROLINA		(803) 772-6760
TEXAS,		
Austin		(512) 346-7830
Dallas		(214) 934-9099
Houston		(713) 785-9001
UTAH		(801) 264-2900

International

BELGIUM, Bruxelles	TEL	(02) 771-91-42
	FAX	(02) 762-37-12
	TLX	846-61028
FRANCE, Paris	TEL	(1) 49-75-10-10
	FAX	(1) 49-75-10-13
	TLX	263282F
WEST GERMANY,		
Hannover area	TEL	(0511) 736085
	FAX	(0511) 721254
	TLX	922850
München	TEL	(089) 4114-0
	FAX	(089) 406490
	TLX	523883
Stuttgart	TEL	(0711) 62 33 77
	FAX	(0711) 625187
	TLX	721882
HONG KONG,	TEL	852-8654525
Wanchai	FAX	852-8654335
	TLX	67955AMDAPHX
ITALY, Milan	TEL	(02) 3390541
		(02) 3533241
	FAX	(02) 3498000
	TLX	843-315286
JAPAN,		
Atsugi	TEL	462-29-8460
	FAX	462-29-8458
Kanagawa	TEL	462-47-2911
	FAX	462-47-1729
Tokyo	TEL	(03) 346-7550
	FAX	(03) 342-5196
	TLX	J24064AMDTKOJ
Osaka	TEL	06-243-3250
	FAX	06-243-3253
KOREA, Seoul	TEL	822-784-0030
	FAX	822-784-8014
LATIN AMERICA,		
Ft. Lauderdale	TEL	(305) 484-8600
	FAX	(305) 485-9736
	TLX	5109554261 AMDFTL
NORWAY, Hovik	TEL	(03) 010156
	FAX	(02) 591959
	TLX	79079
SINGAPORE	TEL	65-3481188
	FAX	65-3480161
	TLX	55650 AMDMMI
SWEDEN,		
Stockholm	TEL	(08) 733 03 50
(Sundbyberg)	FAX	(08) 733 22 85
	TLX	11602
TAIWAN	TEL	886-2-7213393
	FAX	886-2-7723422
	TLX	886-2-7122066
UNITED KINGDOM,		
Manchester area	TEL	(0925) 828008
(Warrington)	FAX	(0925) 827693
	TLX	851-628524
London area	TEL	(0483) 740440
(Woking)	FAX	(0483) 756196
	TLX	851-859103

North American Representatives

CANADA		
Burnaby, B.C. - DAVETEK MARKETING		(604) 430-3680
Calgary, Alberta - DAVETEK MARKETING		(403) 291-4984
Kanata, Ontario - VITEL ELECTRONICS		(613) 592-0060
Mississauga, Ontario - VITEL ELECTRONICS		(416) 676-9720
Lachine, Quebec - VITEL ELECTRONICS		(514) 636-5951
IDAHO		
INTERMOUNTAIN TECH MKTG, INC		(208) 888-6071
ILLINOIS		
HEARTLAND TECH MKTG, INC		(312) 577-9222
INDIANA		
Huntington - ELECTRONIC MARKETING CONSULTANTS, INC		(317) 921-3450
Indianapolis - ELECTRONIC MARKETING CONSULTANTS, INC		(317) 921-3450
IOWA		
LORENZ SALES		(319) 377-4666
KANSAS		
Merriam – LORENZ SALES		(913) 469-1312
Wichita – LORENZ SALES		(316) 721-0500
KENTUCKY		
ELECTRONIC MARKETING CONSULTANTS, INC		(317) 921-3452
MICHIGAN		
Birmingham - MIKE RAICK ASSOCIATES		(313) 644-5040
Holland – COM-TEK SALES, INC		(616) 392-7100
Novi – COM-TEK SALES, INC		(313) 344-1409
MINNESOTA		
Mel Foster Tech. Sales, Inc.		(612) 941-9790
MISSOURI		
LORENZ SALES		(314) 997-4558
NEBRASKA		
LORENZ SALES		(402) 475-4660
NEW MEXICO		
THORSON DESERT STATES		(505) 293-8555
NEW YORK		
East Syracuse – NYCOM, INC		(315) 437-8343
Woodbury – COMPONENT CONSULTANTS, INC		(516) 364-8020
OHIO		
Centerville – DOLFUSS ROOT & CO		(513) 433-6776
Columbus – DOLFUSS ROOT & CO		(614) 885-4844
Strongsville – DOLFUSS ROOT & CO		(216) 899-9370
OREGON		
ELECTRA TECHNICAL SALES, INC		(503) 643-5074
PENNSYLVANIA		
RUSSELL F. CLARK CO.,INC.		(412) 242-9500
PUERTO RICO		
COMP REP ASSOC, INC		(809) 746-6550
UTAH, R² MARKETING		(801) 595-0631
WASHINGTON		
ELECTRA TECHNICAL SALES		(206) 821-7442
WISCONSIN		
HEARTLAND TECH MKTG, INC		(414) 792-0920

Advanced Micro Devices, Inc. 901 Thompson Place, P.O. Box 3453, Sunnyvale, CA 94088, USA
Tel: (408) 732-2400 • TWX: 910-339-9280 • TELEX: 34-6306 • TOLL FREE: (800) 538-8450
APPLICATIONS HOTLINE & LITERATURE ORDERING • TOLL FREE: (800) 222-9323 • (408) 749-5703

10/22/90
Printed in USA