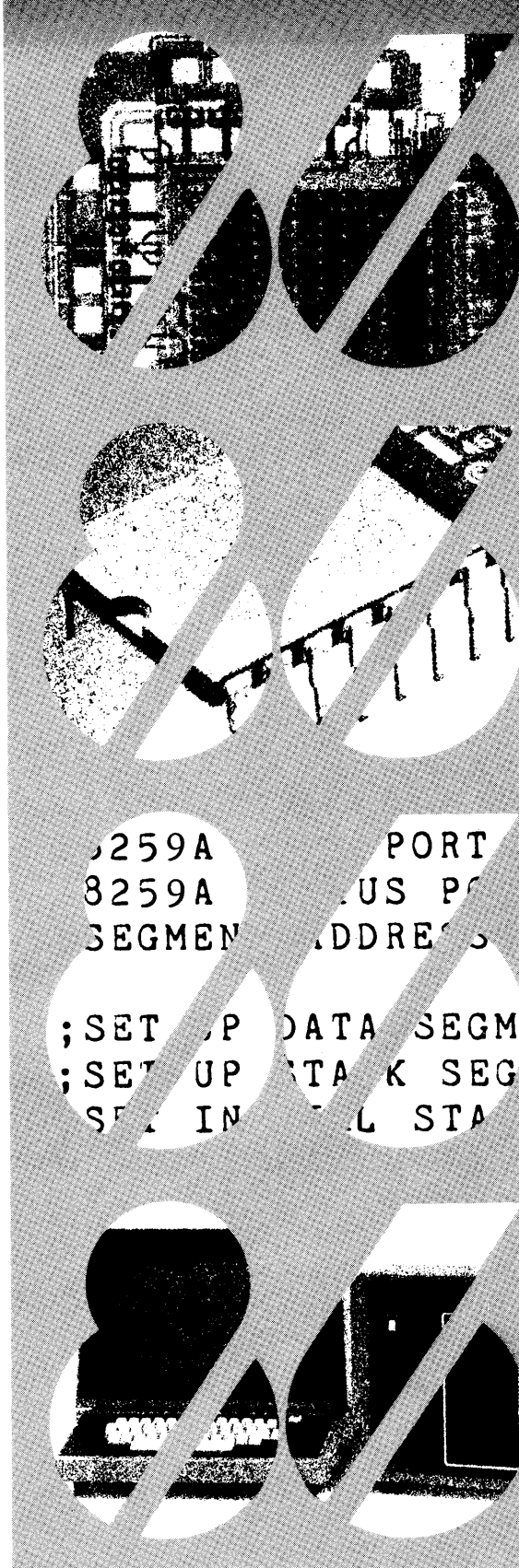
Chapter 2
The 8086 and 8088 Central Processing Units



CHAPTER 2
THE 8086 AND 8088 CENTRAL PROCESSING UNITS

This chapter describes the mainstays of the 8086 microprocessor family: the 8086 and 8088 central processing units (CPUs). The material is divided into ten sections and generally proceeds from hardware to software topics as follows:

1. Processor Overview
2. Processor Architecture
3. Memory
4. Input/Output
5. Multiprocessing Features
6. Processor Control and Monitoring
7. Instruction Set
8. Addressing Modes
9. Programming Facilities
10. Programming Guidelines and Examples

The chapter describes the internal operation of the CPUs in detail. The interaction of the processors with other devices is discussed in functional terms; electrical characteristics, timing, and other information needed to actually interface other devices with the 8086 and 8088 are provided in Chapter 4.

2.1 Processor Overview

The 8086 and 8088 are closely related third-generation microprocessors. The 8088 is designed with an 8-bit external data path to memory and I/O, while the 8086 can transfer 16 bits at a time. In almost every other respect the processors are identical; software written for one CPU will execute on the other without alteration. The chips are contained in standard 40-pin dual in-line packages (figure 2-1) and operate from a single +5V power source.

The 8086 and 8088 are suitable for an exceptionally wide spectrum of microcomputer applications, and this flexibility is one of their most outstanding characteristics. Systems can range from uniprocessor minimal-memory designs implemented with a handful of chips (figure 2-2), to multiprocessor systems with up to a megabyte of memory (figure 2-3).

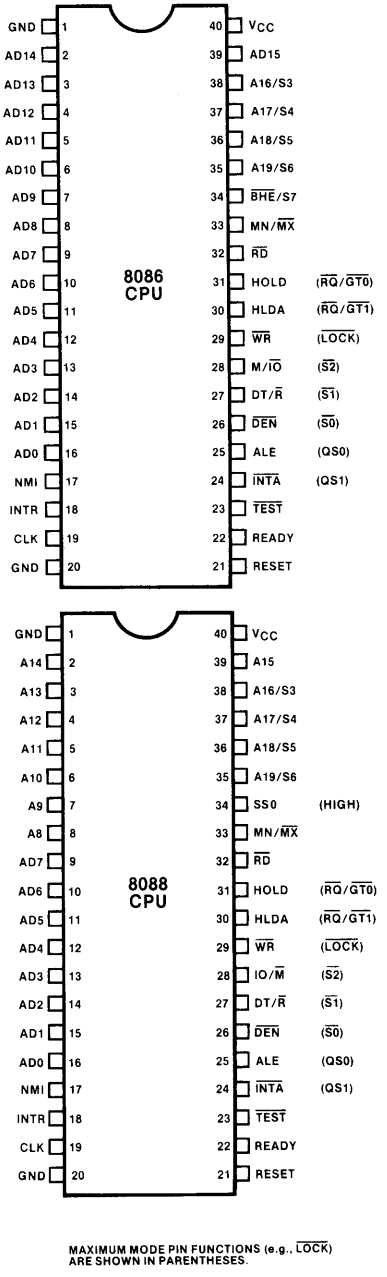


Figure 2-1. 8086 and 8088 Central Processing Units

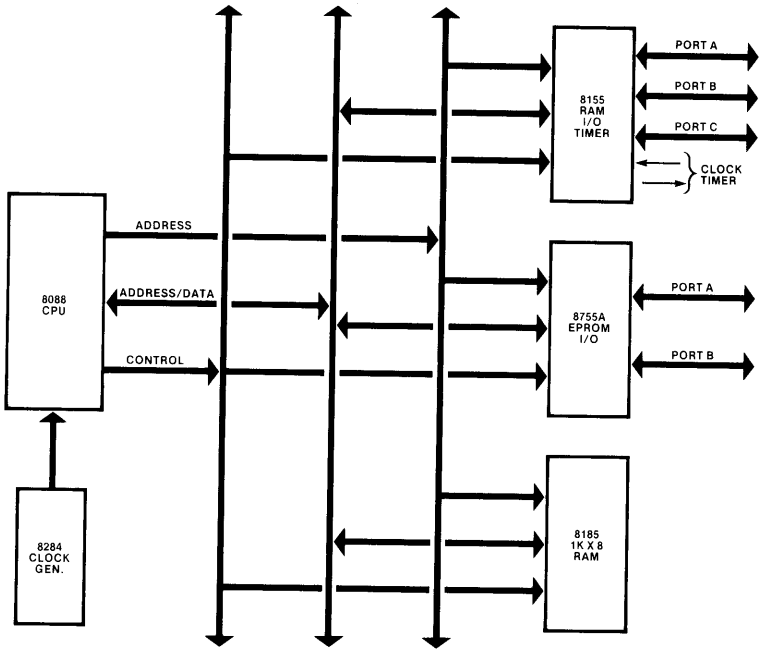


Figure 2-2. Small 8088-Based System

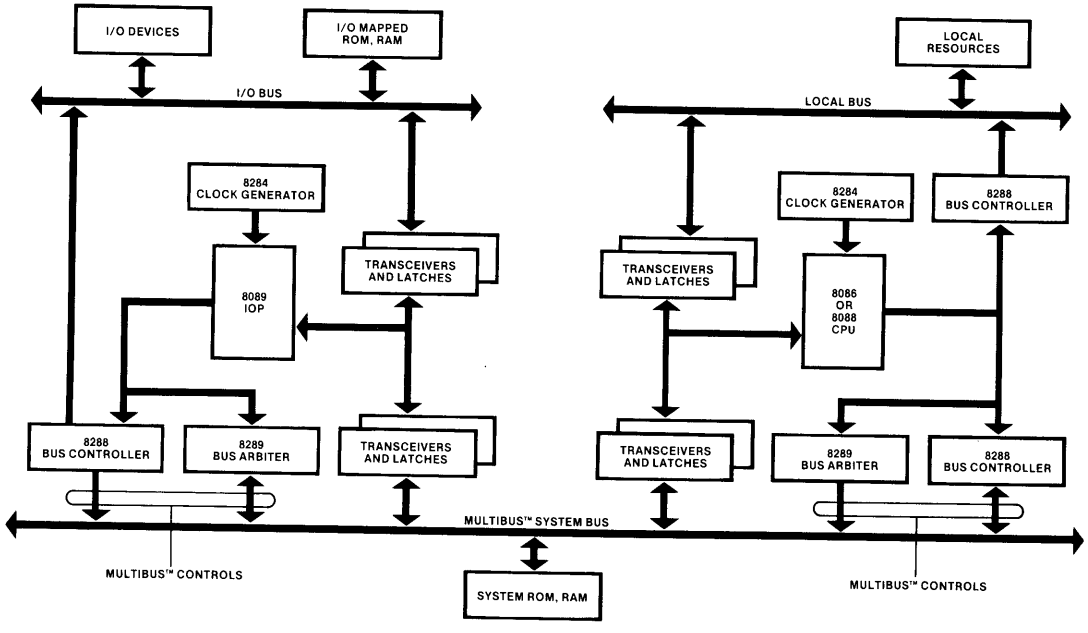


Figure 2-3. 8086/8088/8089 Multiprocessing System

The large application domain of the 8086 and 8088 is made possible primarily by the processors' dual operating modes (minimum and maximum mode) and built-in multiprocessing features. Several of the 40 CPU pins have dual functions that are selected by a strapping pin. Configured in minimum mode, these pins transfer control signals directly to memory and input/output devices. In maximum mode these same pins take on different functions that are helpful in medium to large ystems, especially systems with multiple processors. The control functions assigned to these pins in minimum mode are assumed by a support chip, the 8288 Bus Controller.

The CPUs are designed to operate with the 8089 Input/Output Processor (IOP) and other processors in multiprocessing and distributed processing systems. When used in conjunction with one or more 8089s, the 8086 and 8088 expand the applicability of microprocessors into I/O-intensive data processing systems. Built-in coordinating signals and instructions, and electrical compatibility with Intel's Multibus™ shared bus architecture, simplify and reduce the cost of developing multiple-processor designs.

Both CPUs are substantially more powerful than any microprocessor previously offered by Intel. Actual performance, of course, varies from application to application, but comparisons to the industry standard 2-MHz 8080A are instructive. The 8088 is from four to six times more powerful than the 8080A; the 8086 provides seven to ten times the 8080A's performance (see figure 2-4).

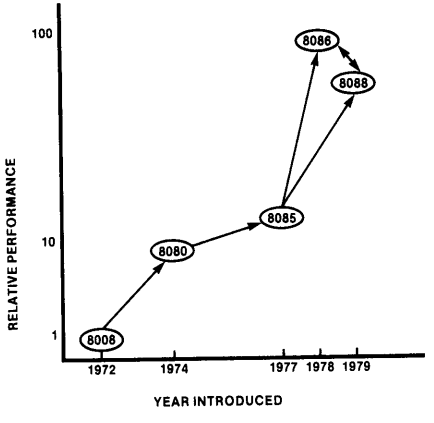


Figure 2-4. Relative Performance of the 8086 and 8088

The 8086's advantage over the 8088 is attributable to its 16-bit external data bus. In applications that manipulate 8-bit quantities extensively, or that are execution-bound, the 8088 can approach to within 10% of the 8086's processing throughput.

The high performance of the 8086 and 8088 is realized by combining a 16-bit internal data path with a pipelined architecture that allows instructions to be prefetched during spare bus cycles. Also contributing to performance is a compact instruction format that enables more instructions to be fetched in a given amount of time.

Software for high-performance 8086 and 8088 systems need not be written in assembly language. The CPUs are designed to provide direct hardware support for programs written in high-level languages such as Intel's PL/M-86. Most high-level languages store variables in memory; the 8086/8088 symmetrical instruction set supports direct operation on memory operands, including operands on the stack. The hardware addressing modes provide efficient, straightforward implementations of based variables, arrays, arrays of structures and other high-level language data constructs. A powerful set of memory-to-memory string operations is available for efficient character data manipulation. Finally, routines with critical performance requirements that cannot be met with PL/M-86 may be written in ASM-86 (the 8086/8088 assembly language) and linked with PL/M-86 code.

While the 8086 and 8088 are totally new designs, they make the most of users' existing investments in systems designed around the 8080/8085 microprocessors. Many of the standard Intel memory, peripheral control and communication chips are compatible with the 8086 and the 8088. Software is developed in the familiar Intellec® Microcomputer Development System environment, and most existing programs, whether written in ASM-80 or PL/M-80, can be directly converted to run on the 8086 and 8088.

2.2 Processor Architecture

Microprocessors generally execute a program by repeatedly cycling through the steps shown below (this description is somewhat simplified):

1. Fetch the next instruction from memory.
2. Read an operand (if required by the instruction).

3. Execute the instruction.
4. Write the result (if required by the instruction).

In previous CPUs, most of these steps have been performed serially, or with only a single bus cycle fetch overlap. The architecture of the 8086 and 8088 CPUs, while performing the same steps, allocates them to two separate processing units within the CPU. The execution unit (EU) executes instructions; the bus interface unit (BIU) fetches instructions, reads operands and writes results. The two units can operate independently of one another and are able, under most circumstances, to extensively overlap instruction fetch with execution. The result is that, in most cases, the time normally required to fetch instructions "disappears" because the EU executes instructions that have already been fetched by the BIU. Figure 2-5 illustrates this overlap and compares it with traditional microprocessor operation. In the example, overlapping reduces the elapsed time required to execute three instructions, and allows two additional instructions to be prefetched as well.

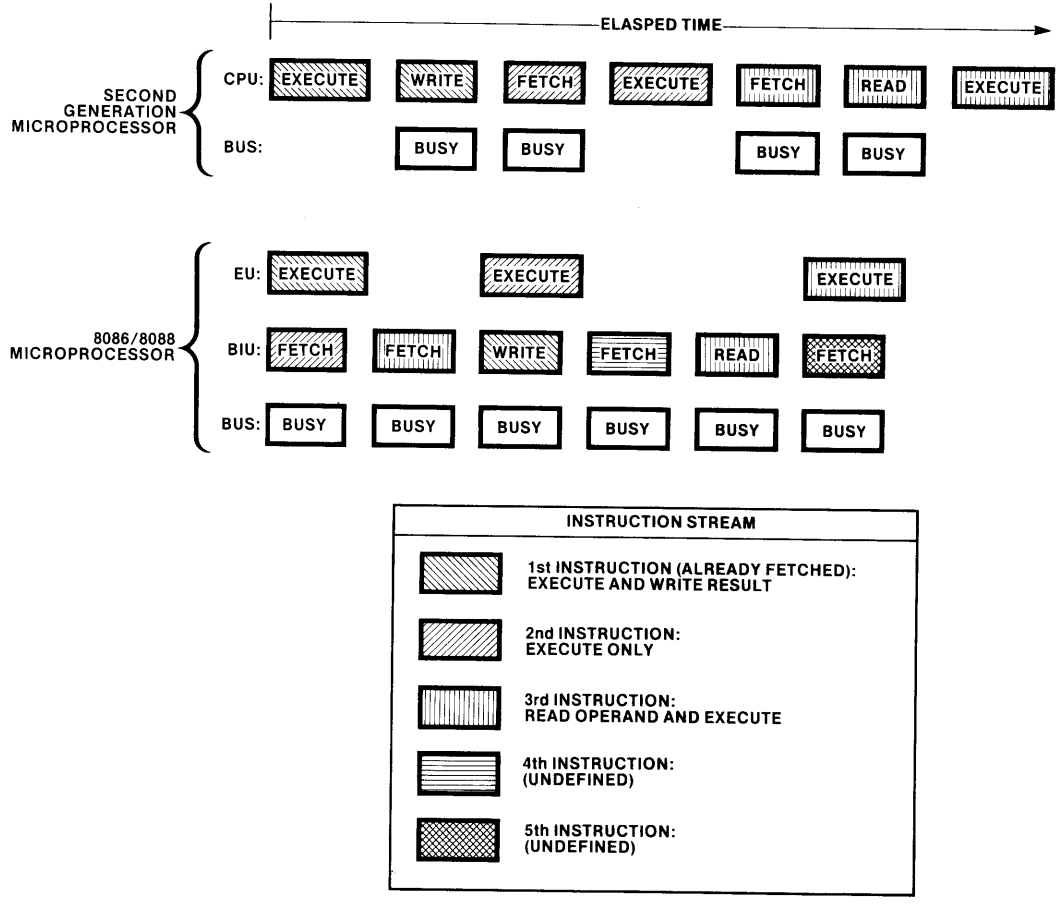


Figure 2-5. Overlapped Instruction Fetch and Execution

Execution Unit

The execution units of the 8086 and 8088 are identical (figure 2-6). A 16-bit arithmetic/logic unit (ALU) in the EU maintains the CPU status and control flags, and manipulates the general registers and instruction operands. All registers and data paths in the EU are 16 bits wide for fast internal transfers.

The EU has no connection to the system bus, the "outside world." It obtains instructions from a queue maintained by the BIU. Likewise, when an instruction requires access to memory or to a peripheral device, the EU requests the BIU to obtain or store the data. All addresses manipulated by the EU are 16 bits wide. The BIU, however, performs an address relocation that gives the EU access to the full megabyte of memory space (see section 2.3).

Bus Interface Unit

The BIUs of the 8086 and 8088 are functionally identical, but are implemented differently to match the structure and performance characteristics of their respective buses.

The BIU performs all bus operations for the EU. Data is transferred between the CPU and memory or I/O devices upon demand from the EU. Sections 2.3 and 2.4 describe the interaction of the BIU with memory and I/O devices.

In addition, during periods when the EU is busy executing instructions, the BIU "looks ahead" and fetches more instructions from memory. The instructions are stored in an internal RAM array called the instruction stream queue. The 8088 instruction queue holds up to four bytes of the instruction stream, while the 8086 queue can store

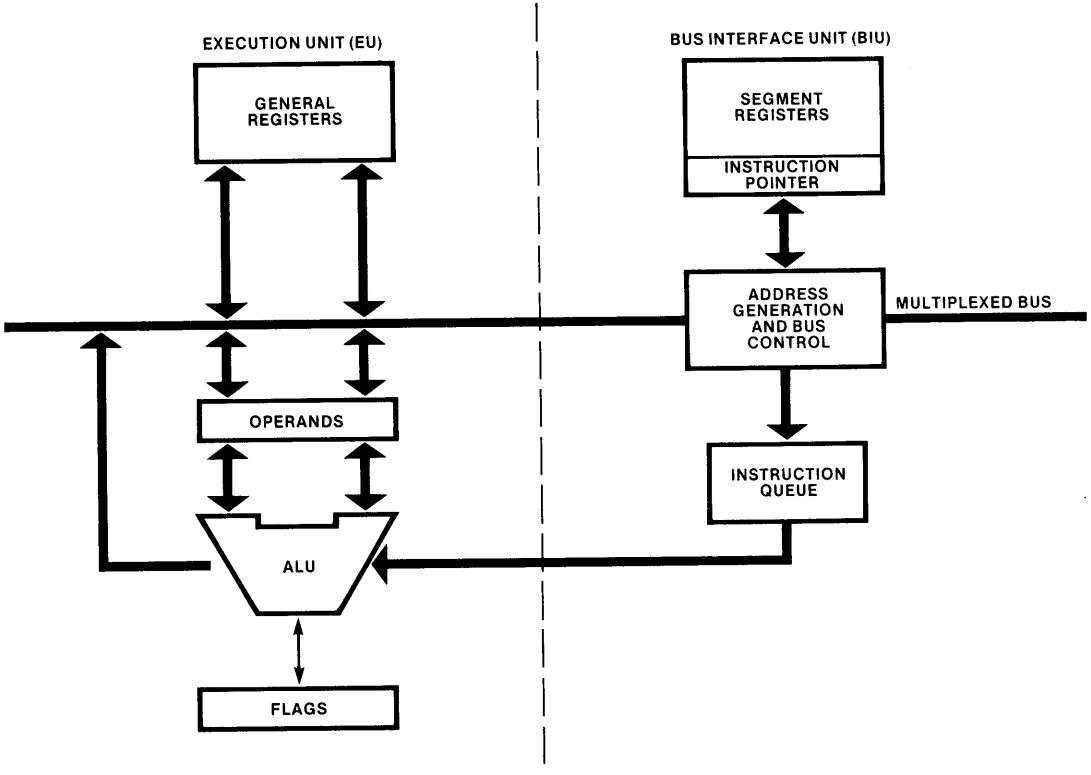


Figure 2-6. Execution and Bus Interface Units (EU and BIU)

up to six instruction bytes. These queue sizes allow the BIU to keep the EU supplied with prefetched instructions under most conditions without monopolizing the system bus. The 8088 BIU fetches another instruction byte whenever one byte in its queue is empty and there is no active request for bus access from the EU. The 8086 BIU operates similarly except that it does not initiate a fetch until there are two empty bytes in its queue. The 8086 BIU normally obtains two instruction bytes per fetch; if a program transfer forces fetching from an odd address, the 8086 BIU automatically reads one byte from the odd address and then resumes fetching two-byte words from the subsequent even addresses.

Under most circumstances the queues contain at least one byte of the instruction stream and the EU does not have to wait for instructions to be fetched. The instructions in the queue are those stored in the memory locations immediately adjacent to and higher than the instruction currently being executed. That is, they are the next logical instructions so long as execution proceeds serially. If the EU executes an instruction that transfers control to another location, the BIU resets the queue, fetches the instruction from the new address, passes it immediately to the EU, and then begins refilling the queue from the new location. In addition, the BIU suspends instruction fetching whenever the EU requests a memory or I/O read or write (except that a fetch already in progress is completed before executing the EU's bus request).

General Registers

Both CPUs have the same complement of eight 16-bit general registers (figure 2-7). The general registers are subdivided into two sets of four registers each: the data registers (sometimes called the H & L group for "high" and "low"), and the pointer and index registers (sometimes called the P & I group).

The data registers are unique in that their upper (high) and lower halves are separately addressable. This means that each data register can be used interchangeably as a 16-bit register, or as two 8-bit registers. The other CPU registers always are accessed as 16-bit units only. The data registers can be used without constraint in most arithmetic and logic operations. In addition,

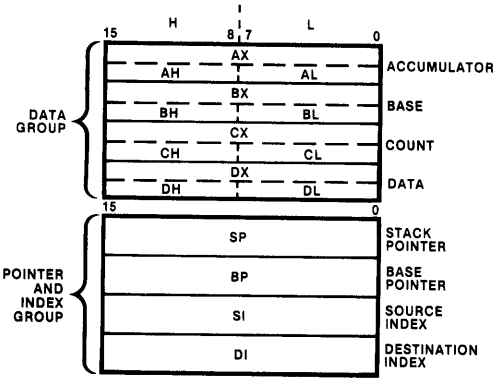


Figure 2-7. General Registers

some instructions use certain registers implicitly (see table 2-1) thus allowing compact yet powerful encoding.

Table 2-1. Implicit Use of General Registers

REGISTER	OPERATIONS
AX	Word Multiply, Word Divide, Word I/O
AL	Byte Multiply, Byte Divide, Byte I/O, Translate, Decimal Arithmetic
AH	Byte Multiply, Byte Divide
BX	Translate
CX	String Operations, Loops
CL	Variable Shift and Rotate
DX	Word Multiply, Word Divide, Indirect I/O
SP	Stack Operations
SI	String Operations
DI	String Operations

The pointer and index registers can also participate in most arithmetic and logic operations. In fact, all eight general registers fit the definition of "accumulator" as used in first and second generation microprocessors. The P & I registers (except for BP) also are used implicitly in some instructions as shown in table 2-1.

Segment Registers

The megabyte of 8086 and 8088 memory space is divided into logical segments of up to 64k bytes each. (Memory segmentation is described in section 2.3.) The CPU has direct access to four segments at a time; their base addresses (starting locations) are contained in the segment registers (see figure 2-8). The CS register points to the current code segment; instructions are fetched from this segment. The SS register points to the current stack segment; stack operations are performed on locations in this segment. The DS register points to the current data segment; it generally contains program variables. The ES register points to the current extra segment, which also is typically used for data storage.

The segment registers are accessible to programs and can be manipulated with several instructions. Good programming practice and consideration of compatibility with future Intel hardware and software products dictate that the segment registers be used in a disciplined fashion. Section 2.10 provides guidelines for segment register use.

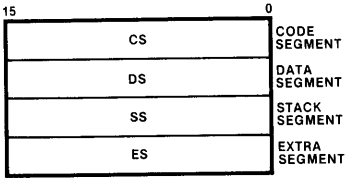


Figure 2-8. Segment Registers

Instruction Pointer

The 16-bit instruction pointer (IP) is analogous to the program counter (PC) in the 8080/8085 CPUs. The instruction pointer is updated by the BIU so that it contains the offset (distance in bytes) of the next instruction from the beginning of the current code segment; i.e., IP points to the next instruction. During normal execution, IP contains the offset of the next instruction to be *fetched* by the BIU; whenever IP is saved on the stack, however, it first is automatically adjusted to point to the next instruction to be *executed*. Programs do not have direct access to the instruction pointer, but instructions cause it to change and to be saved on and restored from the stack.

Flags

The 8086 and 8088 have six 1-bit status flags (figure 2-9) that the EU posts to reflect certain properties of the result of an arithmetic or logic

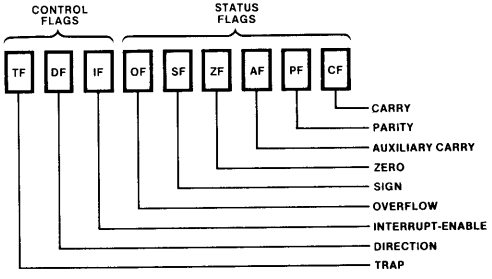


Figure 2-9. Flags

operation. A group of instructions is available that allows a program to alter its execution depending on the state of these flags, that is, on the result of a prior operation. Different instructions affect the status flags differently; in general, however, the flags reflect the following conditions:

1. If AF (the auxiliary carry flag) is set, there has been a carry out of the low nibble into the high nibble or a borrow from the high nibble into the low nibble of an 8-bit quantity (low-order byte of a 16-bit quantity). This flag is used by decimal arithmetic instructions.
2. If CF (the carry flag) is set, there has been a carry out of, or a borrow into, the high-order bit of the result (8- or 16-bit). The flag is used by instructions that add and subtract multibyte numbers. Rotate instructions can also isolate a bit in memory or a register by placing it in the carry flag.
3. If OF (the overflow flag) is set, an arithmetic overflow has occurred; that is, a significant digit has been lost because the size of the result exceeded the capacity of its destination location. An Interrupt On Overflow instruction is available that will generate an interrupt in this situation.

4. If SF (the sign flag) is set, the high-order bit of the result is a 1. Since negative binary numbers are represented in the 8086 and 8088 in standard two's complement notation, SF indicates the sign of the result (0 = positive, 1 = negative).
5. If PF (the parity flag) is set, the result has even parity, an even number of 1-bits. This flag can be used to check for data transmission errors.
6. If ZF (the zero flag) is set, the result of the operation is 0.

Three additional control flags (figure 2-9) can be set and cleared by programs to alter processor operations:

1. Setting DF (the direction flag) causes string instructions to auto-decrement; that is, to process strings from high addresses to low addresses, or from "right to left." Clearing DF causes string instructions to auto-increment, or to process strings from "left to right."
2. Setting IF (the interrupt-enable flag) allows the CPU to recognize external (maskable) interrupt requests. Clearing IF disables these interrupts. IF has no affect on either non-maskable external or internally generated interrupts.
3. Setting TF (the trap flag) puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes instruction by instruction. Section 2.10 contains an example showing the use of TF in a single-step and breakpoint routine.

8080/8085 Registers and Flag Correspondence

The registers, flags and program counter in the 8080/8085 CPUs all have counterparts in the 8086 and 8088 (see figure 2-10). The A register (accumulator) in the 8080/8085 corresponds to the AL register in the 8086 and 8088. The 8080/8085 H & L, B & C, and D & E registers correspond to registers BH, BL, CH, CL, DH and DL, respectively, in the 8086 and 8088. The 8080/8085 SP (stack pointer) and PC (program counter) have their counterparts in the 8086/8088 SP and IP. The AF, CF, PF, SF, and ZF flags are the same in both CPU families. The remaining flags and registers are unique to the 8086 and 8088. This 8080/8085 to 8086 mapping allows most existing 8080/8085 program code to be directly translated into 8086/8088 code.

Mode Selection

Both processors have a strap pin (MN/$\overline{MX}$) that defines the function of eight CPU pins in the 8086 and nine pins in the 8088. Connecting MN/$\overline{MX}$ to +5V places the CPU in minimum mode. In this configuration, which is designed for small systems (roughly one or two boards), the CPU itself provides the bus control signals needed by memory and peripherals. When MN/$\overline{MX}$ is strapped to ground, the CPU is configured in maximum mode. In this configuration the CPU encodes control signals on three lines. An 8288 Bus Controller is added to decode the signals from the CPU and to provide an expanded set of control signals to the rest of the system. The CPU uses the remaining free lines for a new set of signals designed to help coordinate the activities of other processors in the system. Sections 2.5 and 2.6 describe the functions of these signals.

2.3 Memory

The 8086 and 8088 can accommodate up to 1,048,576 bytes of memory in both minimum and maximum mode. This section describes how memory is functionally organized and used. There are substantial differences in the way memory components are actually accessed by the two processors; these differences, which are invisible to programs, are covered in section 4.2, External Memory Addressing.

Storage Organization

From a storage point of view, the 8086 and 8088 memory spaces are organized as identical arrays of 8-bit bytes (see figure 2-11). Instructions, byte data and word data may be freely stored at any byte address without regard for alignment thereby saving memory space by allowing code to be densely packed in memory (see figure 2-12). Odd-addressed (unaligned) word variables, however,

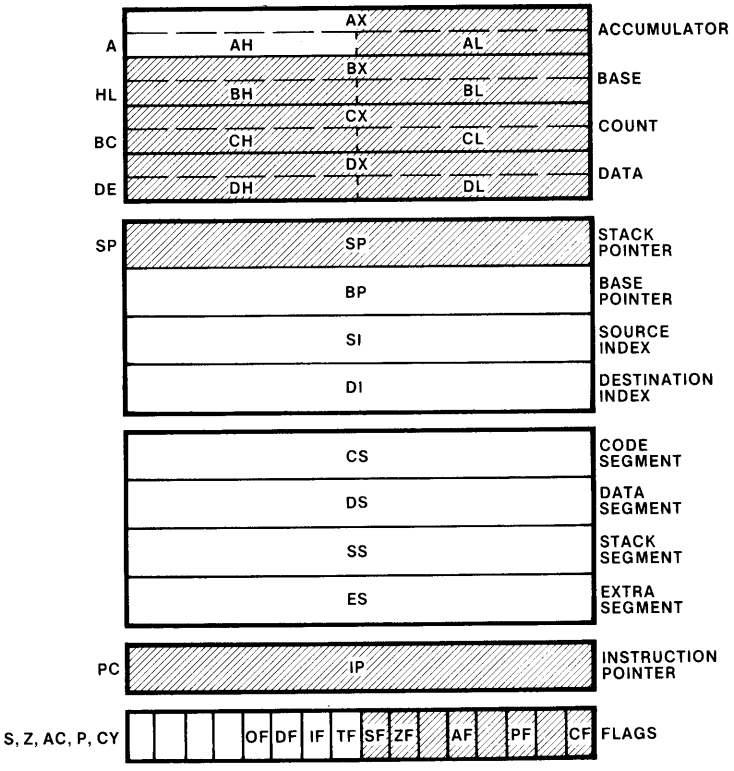


Figure 2-10. 8080/8085 Register Subset (Shaded)

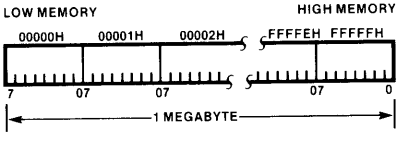


Figure 2-11. Storage Organization

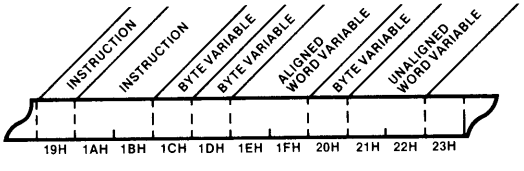
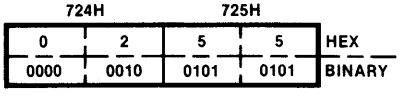


Figure 2-12. Instruction and Variable Storage

do not take advantage of the 8086's ability to transfer 16-bits at a time. Instruction alignment does not materially affect the performance of either processor.

Following Intel convention, word data always is stored with the most-significant byte in the higher memory location (see figure 2-13). Most of the time this storage convention is "invisible" to anyone working with the processors; exceptions may occur when monitoring the system bus or when reading memory dumps.

A special class of data is stored as doublewords; i.e., two consecutive words. These are called pointers and are used to address data and code that are outside the currently-addressable segments. The lower-addressed word of a pointer contains an offset value, and the higher-addressed word contains a segment base address. Each word is stored conventionally with the higher-addressed byte containing the most-significant eight bits of the word (see figure 2-14).



VALUE OF WORD STORED AT 724H: 5502H

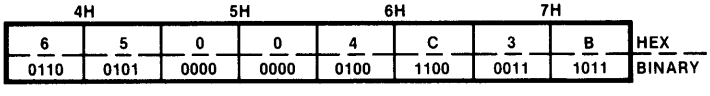
Figure 2-13. Storage of Word Variables

Segmentation

8086 and 8088 programs "view" the megabyte of memory space as a group of segments that are defined by the application. A segment is a logical unit of memory that may be up to 64k bytes long. Each segment is made up of contiguous memory locations and is an independent, separately-addressable unit. Every segment is assigned (by software) a base address, which is its starting location in the memory space. All segments begin on 16-byte memory boundaries. There are no other restrictions on segment locations; segments may be adjacent, disjoint, partially overlapped, or fully overlapped (see figure 2-15). A physical memory location may be mapped into (contained in) one or more logical segments.

The segment registers point to (contain the base address values of) the four currently addressable segments (see figure 2-16). Programs obtain access to code and data in other segments by changing the segment registers to point to the desired segments.

Every application will define and use segments differently. The currently addressable segments provide a generous work space: 64k bytes for code, a 64k byte stack and 128k bytes of data storage. Many applications can be written to simply initialize the segment registers and then forget them. Larger applications should be designed with careful consideration given to segment definition.



VALUE OF POINTER STORED AT 4H:
SEGMENT BASE ADDRESS: 3B4CH
OFFSET: 65H

Figure 2-14. Storage of Pointer Variables

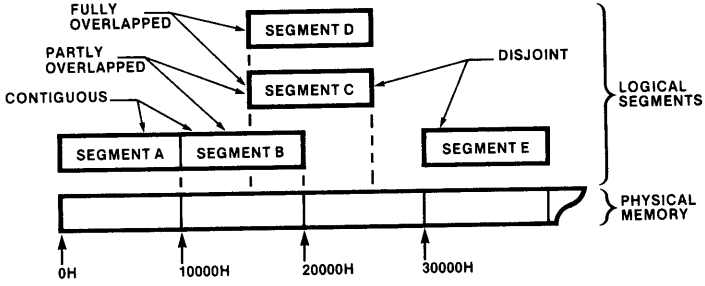


Figure 2-15. Segment Locations in Physical Memory

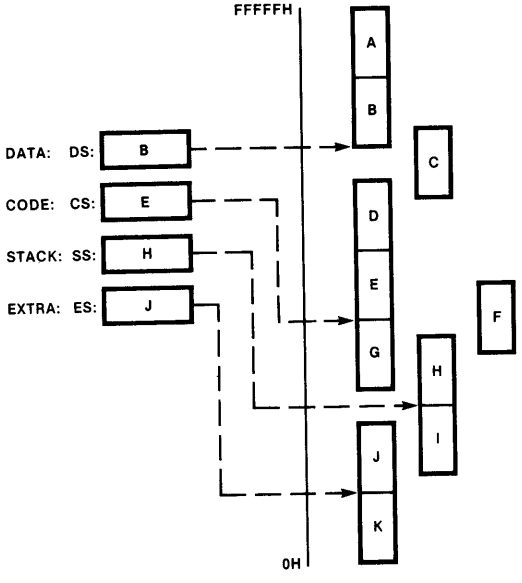


Figure 2-16. Currently Addressable Segments

The segmented structure of the 8086/8088 memory space supports modular software design by discouraging huge, monolithic programs. The segments also can be used to advantage in many programming situations. Take, for example, the case of an editor for several on-line terminals. A 64k text buffer (probably an extra segment) could be assigned to each terminal. A single program could maintain all the buffers by simply changing register ES to point to the buffer of the terminal requiring service.

Physical Address Generation

It is useful to think of every memory location as having two kinds of addresses, physical and logical. A physical address is the 20-bit value that uniquely identifies each byte location in the megabyte memory space. Physical addresses may range from 0H through FFFFFH. All exchanges between the CPU and memory components use this physical address.

Programs deal with logical, rather than physical addresses and allow code to be developed without prior knowledge of where the code is to be located in memory and facilitate dynamic management of memory resources. A logical address consists of a segment base value and an offset value. For any given memory location, the segment base value

locates the first byte of the containing segment and the offset value is the distance, in bytes, of the target location from the beginning of the segment. Segment base and offset values are unsigned 16-bit quantities; the lowest-addressed byte in a segment has an offset of 0. Many different logical addresses can map to the same physical location as shown in figure 2-17. In figure 2-17, physical memory location 2C3H is contained in two different overlapping segments, one beginning at 2B0H and the other at 2C0H.

Whenever the BIU accesses memory—to fetch an instruction or to obtain or store a variable—it generates a physical address from a logical address. This is done by shifting the segment base value four bit positions and adding the offset as illustrated in figure 2-18. Note that this addition process provides for modulo 64k addressing (addresses wrap around from the end of a segment to the beginning of the same segment).

The BIU obtains the logical address of a memory location from different sources depending on the type of reference that is being made (see table 2-2). Instructions always are fetched from the current code segment; IP contains the offset of the target instruction from the beginning of the segment. Stack instructions always operate on the current stack segment; SP contains the offset of the top of the stack. Most variables (memory operands) are assumed to reside in the current data segment, although a program can instruct the BIU to access a variable in one of the other currently addressable segments. The offset of a memory variable is calculated by the EU. This calculation is based on the addressing mode specified in the instruction; the result is called the operand's effective address (EA). Section 2.8 covers addressing modes and effective address calculation in detail.

Strings are addressed differently than other variables. The source operand of a string instruction is assumed to lie in the current data segment, but another currently addressable segment may be specified. Its offset is taken from register SI, the source index register. The destination operand of a string instruction always resides in the current

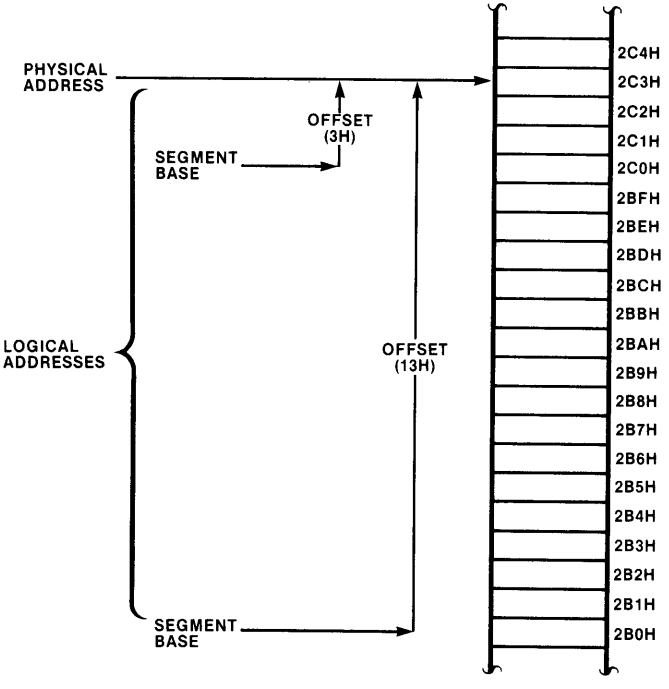


Figure 2-17. Logical and Physical Addresses

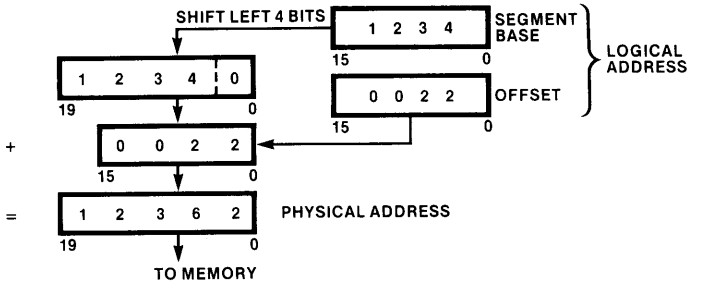


Figure 2-18. Physical Address Generation

Table 2-2. Logical Address Sources

TYPE OF MEMORY REFERENCE	DEFAULT SEGMENT BASE	ALTERNATE SEGMENT BASE	OFFSET
Instruction Fetch	CS	NONE	IP
Stack Operation	SS	NONE	SP
Variable (except following)	DS	CS,ES,SS	Effective Address
String Source	DS	CS,ES,SS	SI
String Destination	ES	NONE	DI
BP Used As Base Register	SS	CS,DS,ES	Effective Address

extra segment; its offset is taken from DI, the destination index register. The string instructions automatically adjust SI and DI as they process the strings one byte or word at a time.

When register BP, the base pointer register, is designated as a base register in an instruction, the variable is assumed to reside in the current stack segment. Register BP thus provides a convenient way to address data on the stack; BP can be used, however, to access data in any of the other currently addressable segments.

In most cases, the BIU's segment assumptions are a convenience to programmers. It is possible, however, for a programmer to explicitly direct the BIU to access a variable in any of the currently addressable segments (the only exception is the destination operand of a string instruction which must be in the extra segment). This is done by preceding an instruction with a segment override prefix. This one-byte machine instruction tells the BIU which segment register to use to access a variable referenced in the following instruction.

Dynamically Relocatable Code

The segmented memory structure of the 8086 and 8088 makes it possible to write programs that are position-independent, or dynamically relocatable. Dynamic relocation allows a multiprogramming or multitasking system to make particularly effective use of available memory. Inactive programs can be written to disk and the space they occupied allocated to other programs. If a disk-resident program is needed later, it can be read back into any available memory location and restarted. Similarly, if a program needs a large contiguous block of storage, and the total amount is available only in nonadjacent fragments, other program segments can be compacted to free up a continuous space. This process is shown graphically in figure 2-19.

In order to be dynamically relocatable, a program must not load or alter its segment registers and must not transfer directly to a location outside the current code segment. In other words, all offsets in the program must be relative to fixed values

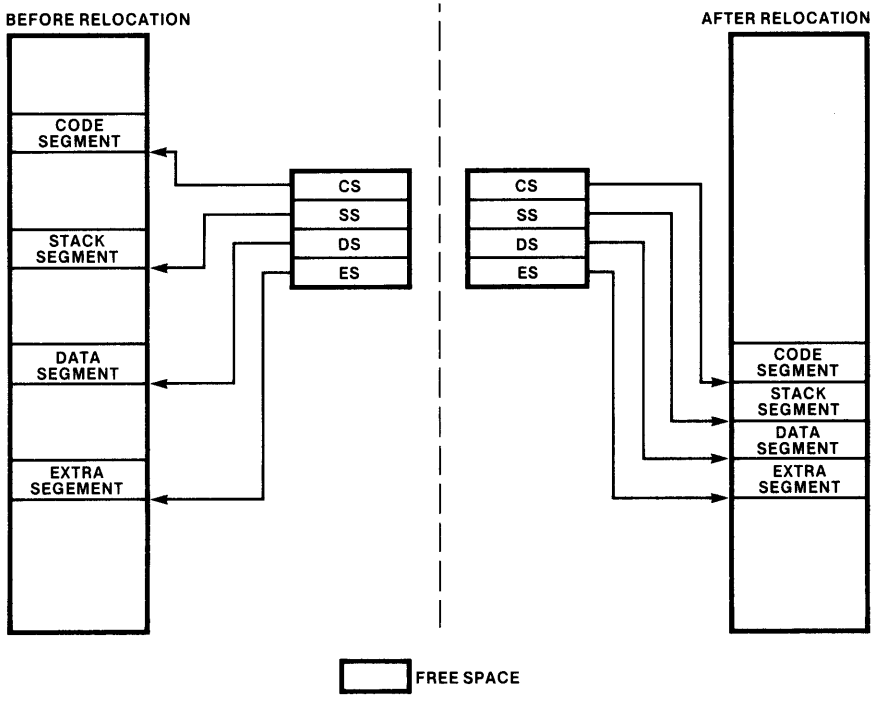


Figure 2-19. Dynamic Code Relocation

contained in the segment registers. This allows the program to be moved anywhere in memory as long as the segment registers are updated to point to the new base addresses. Section 2.10 contains an example that illustrates dynamic code relocation.

Stack Implementation

Stacks in the 8086 and 8088 are implemented in memory and are located by the stack segment register (SS) and the stack pointer register (SP). A system may have an unlimited number of stacks, and a stack may be up to 64k bytes long, the maximum length of a segment. (An attempt to expand a stack beyond 64k bytes overwrites the beginning of the stack.) One stack is directly addressable at a time; this is the current stack, often referred to simply as "the" stack. SS contains the base address of the current stack and SP points to the top of the stack (TOS). In other words, SP contains the offset of the top of the stack from the stack segment's base address. Note, however, that the stack's base address (contained in SS) is not the "bottom" of the stack.

8086 and 8088 stacks are 16 bits wide; instructions that operate on a stack add and remove stack items one word at a time. An item is pushed onto the stack (see figure 2-20) by *decrementing* SP by 2 and writing the item at the new TOS. An item is popped off the stack by copying it from TOS and then *incrementing* SP by 2. In other words, the stack grows *down* in memory toward its base address. Stack operations never move items on the stack, nor do they erase them. The top of the stack changes only as a result of updating the stack pointer.

Dedicated and Reserved Memory Locations

Two areas in extreme low and high memory are dedicated to specific processor functions or are reserved by Intel Corporation for use by Intel

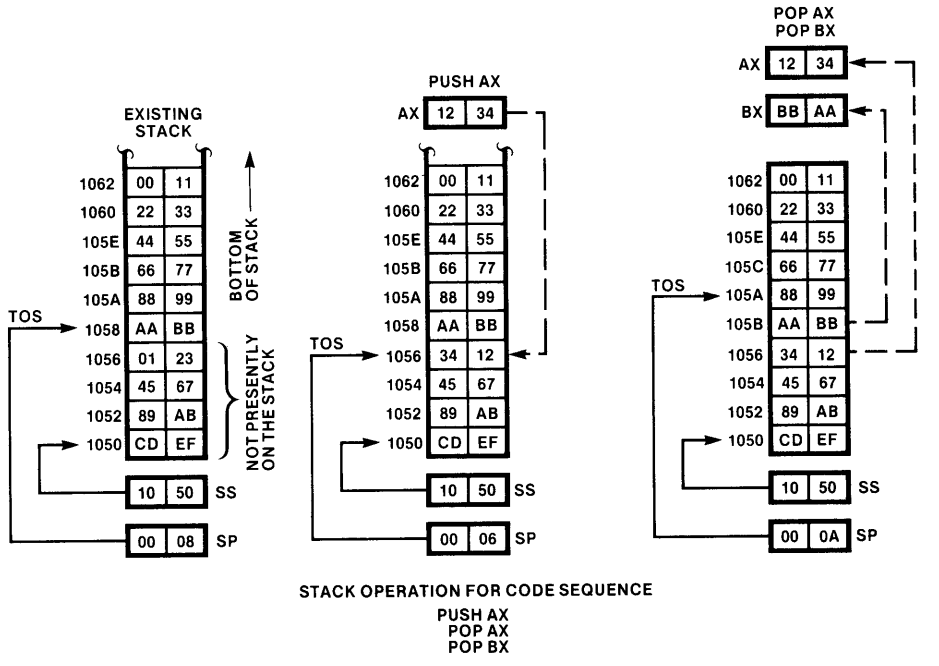


Figure 2-20. Stack Operation

hardware and software products. As shown in figure 2-21, the location are: 0H throgh 7FH (128 bytes) and FFFF0H through FFFFFH (16 bytes). These areas are used for interrupt and system reset processing 8086 and 8088 application systems should not use these areas for any other purpose. Doing so may make these systems incompatible with future Intel products.

8086/8088 Memory Access Differences

The 8086 can access either 8 or 16 bits of memory at a time. If an instruction refers to a word variable and that variable is located at an even-numbered address, the 8086 accesses the complete word in one bus cycle. If the word is located at an odd-numbered address, the 8086 accesses the word one byte at a time in two consecutive bus cycles.

To maximize throughput in 8086-based systems, 16-bit data should be stored at even addresses (should be word-aligned). This is particularly true of stacks. Unaligned stacks can slow a system's response to interrupts. Nevertheless, except for the performance penalty, word alignment is totally transparent to software. This allows maximum data packing where memory space is constrained.

The 8086 always fetches the instruction stream in words from even addresses except that the first fetch after a program transfer to an odd address obtains a byte. The instruction stream is disassembled inside the processor and instruction alignment will not materially affect the performance of most systems.

The 8088 always accesses memory in bytes. Word operands are accessed in two bus cycles regardless of their alignment. Instructions also are fetched one byte at a time. Although alignment of word operands does not affect the performance of the 8088, locating 16-bit data on even addresses will insure maximum throughput if the system is ever transferred to an 8086.

2.4 Input/Output

The 8086 and 8088 have a versatile set of input/output facilities. Both processors provide a large I/O space that is separate from the memory

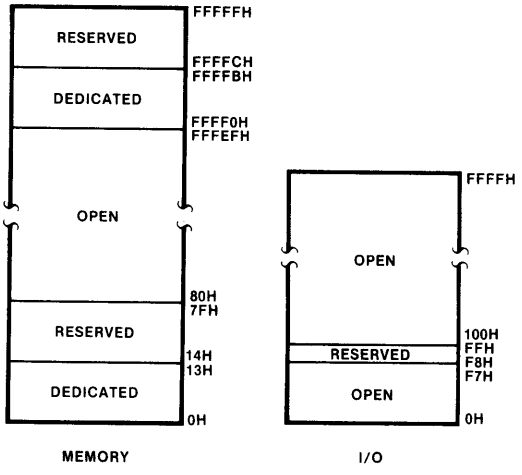


Figure 2-21. Reserved and Dedicated Memory and I/O Locations

space, and instructions that transfer data between the CPU and devices located in the I/O space. I/O devices also may be placed in the memory space to bring the power of the full instruction set and addressing modes to input/output processing. For high-speed transfers, the CPUs may be used with traditional direct memory access controllers or the 8089 Input/Output Processor.

Input/Output Space

The 8086/8088 I/O space can accommodate up to 64k 8-bit ports or up to 32k 16-bit ports. The IN and OUT (input and output) instructions transfer data between the accumulator (AL for byte transfers, AX for word transfers) and ports located in the I/O space.

The I/O space is not segmented; to access a port, the BIU simply places the port address (0-64k) on the lower 16 lines of the address bus. Different forms of the I/O instructions allow the address to be specified as a fixed value in the instruction or as a variable taken from register DX.

Restricted I/O Locations

Locations F8H through FFH (eight of the 64k locations) in the I/O space are reserved by Intel Corporation for use by future Intel hardware and software products. Using these locations for any other purpose may inhibit compatibility with future Intel products.

8086/8088 I/O Access Differences

The 8086 can transfer either 8 or 16 bits at a time to a device located in the I/O space. A 16-bit device should be located at an even address so that the word will be transferred in a single bus cycle. An 8-bit device may be located at either an even or odd address; however, the internal registers in a given device must be assigned all-even or all-odd addresses.

The 8088 transfers one byte per bus cycle. If a 16-bit device is used in the 8088 I/O space, it must be capable of transferring words in the same fashion, i.e., eight bits at a time in two bus cycles. (The 8089 Input/Output Processor can provide a straightforward interface between the 8088 and a 16-bit I/O device.) An 8-bit device may be located at odd or even addresses in the 8088 I/O space and internal registers may be assigned consecutive addresses (e.g., 1H, 2H, 3H). Assigning all-odd or all-even addresses to these registers, however, will simplify transferring the system to an 8086 CPU.

Memory-Mapped I/O

I/O devices also may be placed in the 8086/8088 memory space. As long as the devices respond like memory components, the CPU does not know the difference.

Memory-mapped I/O provides additional programming flexibility. Any instruction that references memory may be used to access an I/O port located in the memory space. For example, the MOV (move) instruction can transfer data between any 8086/8088 register and a port, or the AND, OR and TEST instructions may be used to manipulate bits in I/O device registers. In addition, memory-mapped I/O can take advantage of the 8086/8088 memory addressing modes. A group of terminals, for example, could be treated as an array in memory with an index register

selecting a terminal in the array. Section 2.10 provides examples of using the instruction set and addressing modes with memory-mapped I/O.

Of course, a price must be paid for the added programming flexibility that memory-mapped I/O provides. Dedicating part of the memory space to I/O devices reduces the number of addresses available for memory, although with a megabyte of memory space this should rarely be a constraint. Memory reference instructions also take longer to execute and are somewhat less compact than the simpler IN and OUT instructions.

Direct Memory Access

When configured in minimum mode, the 8086 and 8088 provide HOLD (hold) and HLDA (hold acknowledge) signals that are compatible with traditional DMA controllers such as the 8257 and 8237. A DMA controller can request use of the bus for direct transfer of data between an I/O device and memory by activating HOLD. The CPU will complete the current bus cycle, if one is in progress, and then issue HLDA, granting the bus to the DMA controller. The CPU will not attempt to use the bus until HOLD goes inactive.

The 8086 addresses memory that is physically organized in two separate banks, one containing even-addressed bytes and one containing odd-addressed bytes. An 8-bit DMA controller must alternately select these banks to access logically adjacent bytes in memory. The 8089 provides a simple way to interface a high-speed 8-bit device to an 8086-based system (see Chapter 3).

8089 Input/Output Processor (IOP)

The 8086 and 8088 are designed to be used with the 8089 in high-performance I/O applications. The 8089 conceptually resembles a microprocessor with two DMA channels and an instruction set specifically tailored for I/O operations. Unlike simple DMA controllers, the 8089 can service I/O devices directly, removing this task from the CPU. In addition, it can transfer data on its own bus or on the system bus, can match 8- or 16-bit peripherals to 8- or 16-bit buses, and can transfer data from memory to memory and from I/O device to I/O device. Chapter 3 describes the 8089 in detail.

2.5 Multiprocessing Features

As microprocessor prices have declined, multiprocessing (using two or more coordinated processors in a system) has become an increasingly attractive design alternative. Performance can be substantially improved by distributing system tasks among separate, concurrently executing processors. In addition, multiprocessing encourages a modular approach to design, usually resulting in systems that are more easily maintained and enhanced. For example, figure 2-22 shows a multiprocessor system in which I/O activities have been delegated to an 8089 IOP. Should an I/O device in the system be changed (e.g., a hard disk substituted for a floppy), the impact of the modification is confined to the I/O subsystem and is transparent to the CPU and to the application software.

The 8086 and 8088 are designed for the multiprocessing environment. They have built-in features that help solve the coordination problems that have discouraged multiprocessing system development in the past.

Bus Lock

When configured in maximum mode, the 8086 and 8088 provide the $\overline{\text{LOCK}}$ (bus lock) signal. The BIU activates $\overline{\text{LOCK}}$ when the EU executes the one-byte LOCK prefix instruction. The $\overline{\text{LOCK}}$ signal remains active throughout execution of the instruction that follows the LOCK prefix. Interrupts are *not* affected by the LOCK prefix. If another processor requests use of the bus (via the request/grant lines, which are discussed shortly), the CPU records the request, but does not honor it until execution of the locked instruction has been completed.

Note that the $\overline{\text{LOCK}}$ signal remains active for the duration of a *single* instruction. If two consecutive instructions are each preceded by a LOCK prefix, there will still be an unlocked period between these instructions. In the case of a locked repeated string instruction, $\overline{\text{LOCK}}$ does remain active for the duration of the block operation.

When the 8086 or 8088 is configured in minimum mode, the $\overline{\text{LOCK}}$ signal is not available. The LOCK prefix can be used, however, to delay the

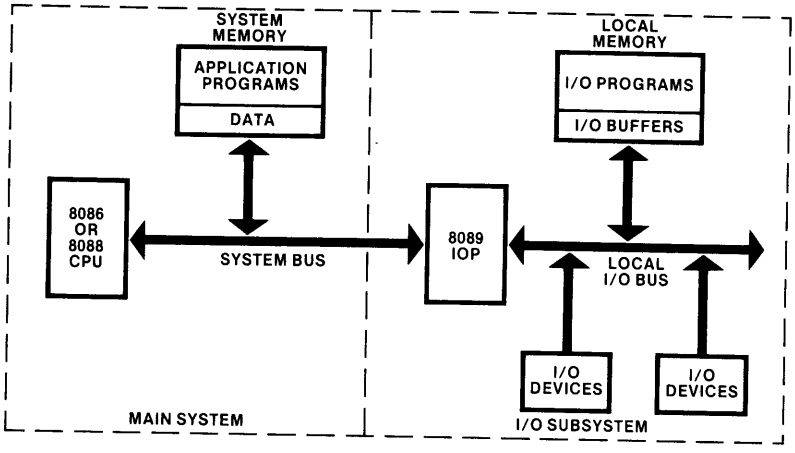


Figure 2-22. Multiprocessing System

generation of an HLDA response to a HOLD request until execution of the locked instruction is completed.

The $\overline{\text{LOCK}}$ signal provides information only. It is the responsibility of other processors on the shared bus to not attempt to obtain the bus while $\overline{\text{LOCK}}$ is active. If the system uses 8289 Bus Arbiters to control access to the shared bus, the 8289's accept $\overline{\text{LOCK}}$ as an input and do not relinquish the bus while this signal is active.

$\overline{\text{LOCK}}$ may be used in multiprocessing systems to coordinate access to a common resource, such as a buffer or a pointer. If access to the resource is not controlled, one processor can read an erroneous value from the resource when another processor is updating it (see figure 2-23).

Access can be controlled (see figure 2-24) by using the LOCK prefix in conjunction with the XCHG (exchange register with memory) instruction. The basis for controlling access to a given resource is a semaphore, a software-settable flag or switch that indicates whether the resource is "available" (semaphore=0) or "busy" (semaphore=1). Processors that share the bus agree by convention not to use the resource unless the semaphore indicates that it is available. They likewise agree to set the semaphore when they are using the resource and to clear it when they are finished.

The XCHG instruction can obtain the current value of the semaphore and set it to "busy" in a single instruction. The instruction, however, requires two bus cycles to swap 8-bit values. It is possible for another processor to obtain the bus between these two cycles and to gain access to the partially-updated semaphore. This can be prevented by preceding the XCHG instruction with a LOCK prefix, as illustrated in figure 2-25. The bus lock establishes control over access to the semaphore and thus to the shared resource.

WAIT and $\overline{\text{TEST}}$

The 8086 and 8088 (in either maximum or minimum mode) can be synchronized to an external event with the WAIT (wait for $\overline{\text{TEST}}$) instruction and the $\overline{\text{TEST}}$ input signal. When the EU executes a WAIT instruction, the result depends on the state of the $\overline{\text{TEST}}$ input line. If $\overline{\text{TEST}}$ is inactive, the processor enters an idle state and repeatedly retests the $\overline{\text{TEST}}$ line at five-clock intervals. If $\overline{\text{TEST}}$ is active, execution continues with the instruction following the WAIT.

BUS CYCLE	SHARED POINTER IN MEMORY	PROCESSOR ACTIVITIES
0	05, 22 | 4C, 1B	
1	C2, 59 | 4C, 1B	"A" UPDATES 1 WORD
2	C2, 59 | 4C, 1B	"B" READS PARTIALLY UPDATED VALUE
3	C2, 59 | 31, 05	"A" COMPLETES UPDATE

Figure 2-23. Uncontrolled Access to Shared Resource

Escape

The ESC (escape) instruction provides a way for another processor to obtain an instruction and/or a memory operand from an 8086/8088 program. When used in conjunction with WAIT and TEST, ESC can initiate a "subroutine" that executes concurrently in another processor (see figure 2-26).

Six bits in the ESC instruction may be specified by the programmer when the instruction is written. By monitoring the 8086/8088 bus and control lines, another processor can capture the ESC instruction when it is fetched by the BIU. The six bits may then direct the external processor to perform some predefined activity.

If the 8086/8088 is configured in maximum mode, the external processor, having determined that an ESC has been fetched, can monitor QS0

BUS CYCLE	SEMAPHORE	SHARED POINTER IN MEMORY	PROCESSOR ACTIVITIES
0	0	05, 22 | 4C, 1B	
1	1	05, 22 | 4C, 1B	"A" OBTAINS EXCLUSIVE USE
2	1	C2, 59 | 4C, 1B	"A" UPDATES 1 WORD
3	1	C2, 59 | 4C, 1B	"B" TESTS SEMAPHORE AND WAITS
4	1	C2, 59 | 31, 05	"A" COMPLETES UPDATE
5	1	C2, 59 | 31, 05	"B" TESTS SEMAPHORE AND WAITS
6	0	C2, 59 | 31, 05	"A" RELEASES RESOURCE
7	1	C2, 59 | 31, 05	"B" OBTAINS EXCLUSIVE USE
8	1	C2, 59 | 31, 05	"B" READS UPDATED VALUE
9	0	C2, 59 | 31, 05	"B" RELEASES RESOURCE

Figure 2-24. Controlled Access to Shared Resource

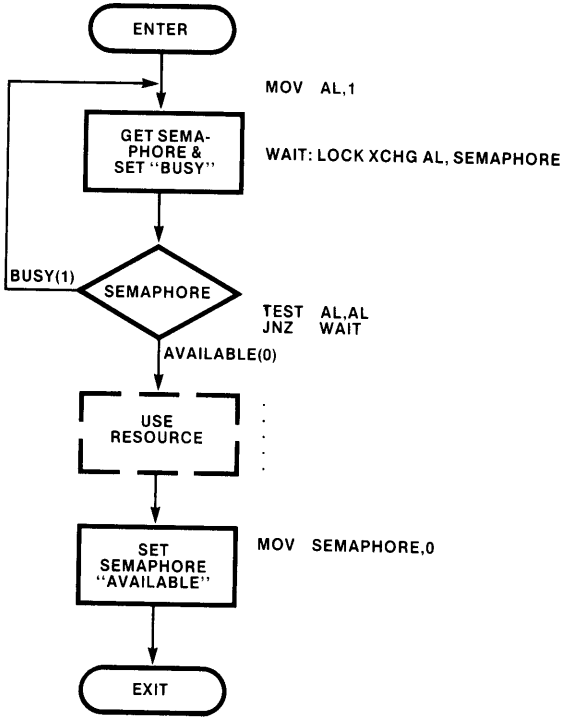


Figure 2-25. Using XCHG and LOCK

and QS1 (the queue status lines, discussed in section 2.6) and determine when the ESC instruction is executed. If the instruction references memory the external processor can then monitor the bus and capture the operand's physical address and/or the operand itself.

Note that fetching an ESC instruction is not tantamount to executing it. The ESC may be preceded by a jump that causes the queue to be reinitialized. This event also can be determined from the queue status lines.

Request/Grant Lines

When the 8086 or 8088 is configured in maximum mode, the HOLD and HLDA lines evolve into two more sophisticated signals called $\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$. These are bidirectional lines that can be used to share a local bus between an 8086 or 8088 and two other processors via a handshake sequence.

The request/grant sequence is a three-phase cycle: request, grant and release. First, the processor desiring the bus pulses a request/grant line. The CPU returns a pulse on the same line indicating that it is entering the "hold acknowledge" state and is relinquishing the bus. The BIU is logically disconnected from the bus during this period. The

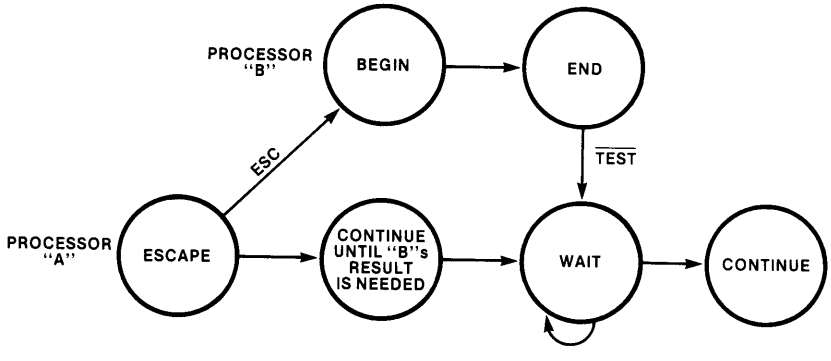


Figure 2-26. Using ESC with WAIT and TEST

EU, however, will continue to execute instructions until an instruction requires bus access or the queue is emptied, whichever occurs first. When the other processor has finished with the bus, it sends a final pulse to the 8086/8088 indicating that the request has ended and that the CPU may reclaim the bus.

$\overline{RQ}/\overline{GT0}$ has higher priority than $\overline{RQ}/\overline{GT1}$. If requests arrive simultaneously on both lines, the grant goes to the processor on $\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$ is acknowledged after the bus has been returned to the CPU. If, however, a request arrives on $\overline{RQ}/\overline{GT0}$ while the CPU is processing a prior request on $\overline{RQ}/\overline{GT1}$, the second request is not honored until the processor on $\overline{RQ}/\overline{GT1}$ releases the bus.

Multibus™ Architecture

Intel has designed a general-purpose multiprocessing bus called the Multibus. This is the standard design used in iSBC™ single-board microcomputer products. Many other manufacturers offer products that are compatible with the Multibus architecture as well. When the 8086 and 8088 are configured in maximum mode, the 8288 Bus Controller outputs signals that are electrically compatible with the Multibus protocol. Designers of multiprocessing systems may want to consider using the Multibus architecture in the design of their products to reduce development cost and time, and to obtain compatibility with the wide variety of boards available in the iSBC product line.

The Multibus architecture provides a versatile communications channel that can be used to coordinate a wide variety of computing modules (see figure 2-27). Modules in a Multibus system are designated as masters or slaves. Masters may obtain use of the bus and initiate data transfers on it. Slaves are the objects of data transfers only. The Multibus architecture allows both 8- and 16-bit masters to be intermixed in a system. In addition to 16 data lines, the bus design provides 20 address lines, eight multilevel interrupt lines, and control and arbitration lines. An auxiliary power bus also is provided to route standby power to memories if the normal supply fails.

The Multibus architecture maintains its own clock, independent of the clocks of the modules it links together. This allows different speed masters to share the bus and allows masters to operate asynchronously with respect to each other. The arbitration logic of the bus permit slow-speed masters to compete equably for use of the bus. Once a module has obtained the bus, however, transfer speeds are dependent only on the capabilities of the transmitting and receiving modules. Finally, the Multibus standard defines the form factors and physical requirements of modules that communicate on this bus. For a complete description of the Multibus architec-

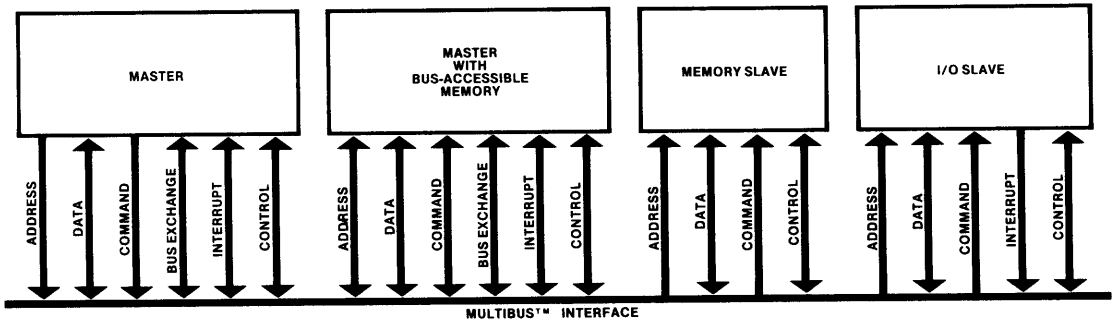


Figure 2-27. Multibus™-Based System

ture, refer to the Intel Multibus Specification (document number 9800683) and Application Note 28A, "Intel Multibus Interfacing."

8289 Bus Arbiter

Multiprocessor systems require a means of coordinating the processors' use of the shared bus. The 8289 Bus Arbiter works in conjunction with the 8288 Bus Controller to provide this control for 8086- and 8088-based systems. It is compatible with the Multibus architecture and can be used in other shared-bus designs as well.

The 8289 eliminates race conditions, resolves bus contention and matches processors operating asynchronously with respect to each other. Each processor on the bus is assigned a different priority. When simultaneous requests for the bus arrive, the 8289 resolves the contention and grants the bus to the processor with the highest priority; three different prioritizing techniques may be used. Chapter 4 discusses the 8289 in more detail.

2.6 Processor Control and Monitoring

Interrupts

The 8086 and 8088 have a simple and versatile interrupt system. Every interrupt is assigned a type code that identifies it to the CPU. The 8086 and 8088 can handle up to 256 different interrupt types. Interrupts may be initiated by devices external to the CPU; in addition, they also may be triggered by software interrupt instructions and, under certain conditions, by the CPU itself (see figure 2-28). Figure 2-29 illustrates the basic response of the 8086 and 8088 to an interrupt. The next sections elaborate on the information presented in this drawing.

External Interrupts

The 8086 and 8088 have two lines that external devices may use to signal interrupts (INTR and NMI). The INTR (Interrupt Request) line is usually driven by an Intel® 8259A Programmable Interrupt Controller (PIC), which is in turn connected to the devices that need interrupt services. The 8259A is a very flexible circuit that is controlled by software commands from the 8086 or 8088 (the PIC appears as a set of I/O ports to the software). Its main job is to accept interrupt requests from the devices attached to it, determine which requesting device has the highest priority, and then activate the 8086/8088 INTR line if the selected device has higher priority than the device currently being serviced (if there is one).

When INTR is active, the CPU takes different action depending on the state of the interrupt-enable flag (IF). No action takes place, however, until the currently-executing instruction has been

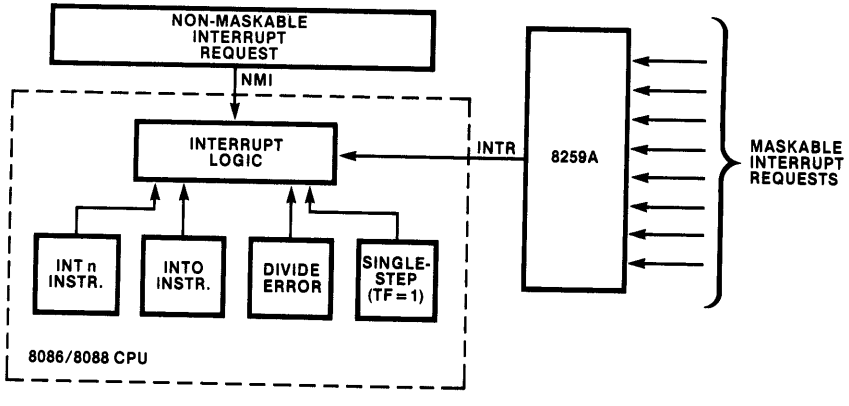


Figure 2-28. Interrupt Sources

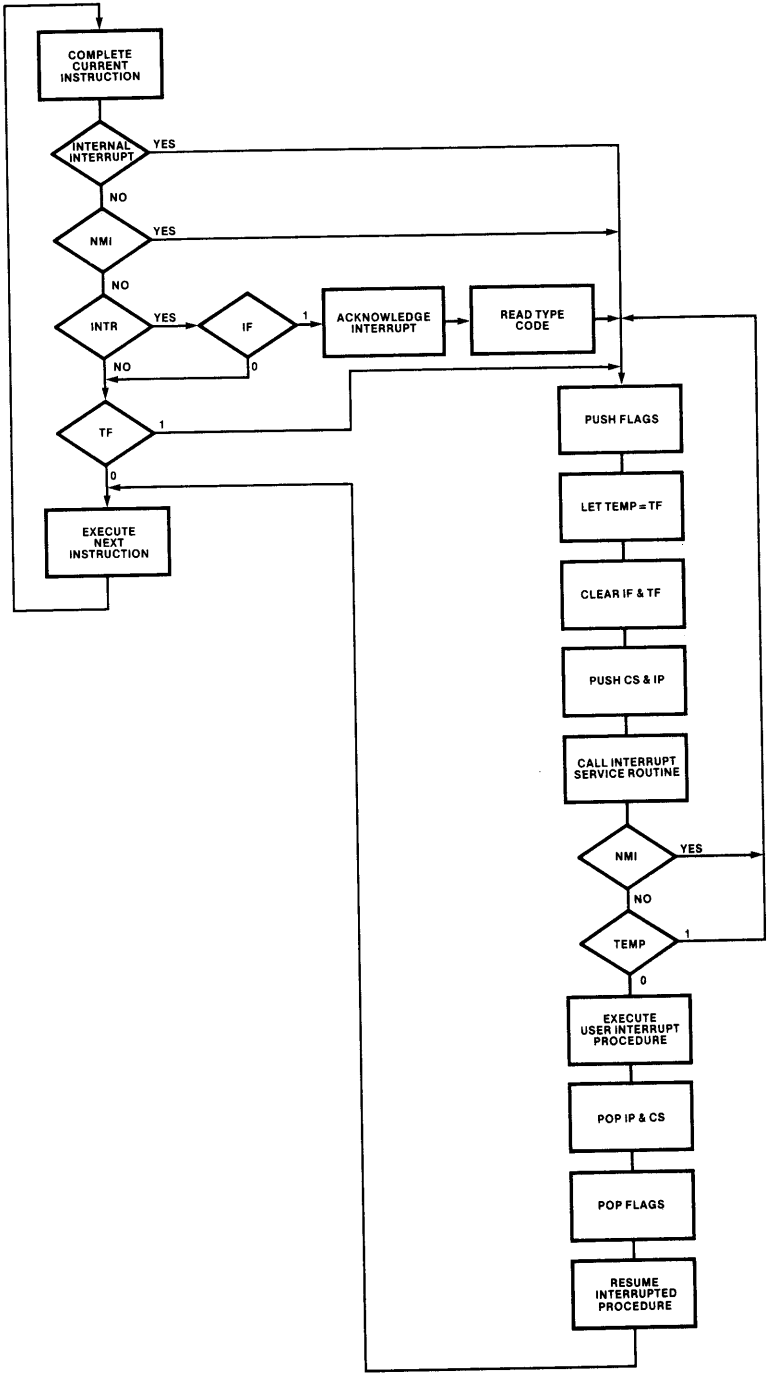


Figure 2-29. Interrupt Processing Sequence

completed.* Then, if IF is clear (meaning that interrupts signaled on INTR are masked or disabled), the CPU ignores the interrupt request and processes the next instruction. The INTR signal is not latched by the CPU, so it must be held active until a response is received or the request is withdrawn. If interrupts on INTR are enabled (if IF is set), then the CPU recognizes the interrupt request and processes it. Interrupt requests arriving on INTR can be enabled by executing an STI (set interrupt-enable flag) instruction, and disabled by executing a CLI (clear interrupt-enable flag) instruction. They also may be selectively masked (some types enabled, some disabled) by writing commands to the 8259A. It should be noted that in order to reduce the likelihood of excessive stack buildup, the STI and IRET instructions will reenable interrupts only after the end of the following instruction.

The CPU acknowledges the interrupt request by executing two consecutive interrupt acknowledge (INTA) bus cycles. If a bus hold request arrives (via the HOLD or request/grant lines) during the INTA cycles, it is not honored until the cycles have been completed. In addition, if the CPU is configured in maximum mode, it activates the $\overline{\text{LOCK}}$ signal during these cycles to indicate to other processors that they should not attempt to obtain the bus. The first cycle signals the 8259A that the request has been honored. During the second INTA cycle, the 8259A responds by placing a byte on the data bus that contains the interrupt type (0-255) associated with the device requesting service. (The type assignment is made when the 8259A is initialized by software in the 8086 or 8088.) The CPU reads this type code and uses it to call the corresponding interrupt procedure.

An external interrupt request also may arrive on another CPU line, NMI (non-maskable interrupt). This line is edge-triggered (INTR is level-triggered) and is generally used to signal the CPU of a "catastrophic" event, such as the imminent loss of power, memory error detection or bus parity error. Interrupt requests arriving on NMI cannot be disabled, are latched by the CPU, and have higher priority than an interrupt request on INTR. If an interrupt request arrives on both lines during the execution of an instruction, NMI will be recognized first. Non-maskable interrupts are predefined as type 2; the processor does not need to be supplied with a type code to call the NMI procedure, and it does not run the INTA bus cycles in response to a request on NMI.

The time required for the CPU to recognize an external interrupt request (interrupt latency) depends on how many clock periods remain in the execution of the current instruction. On the average, the longest latency occurs when a multiplication, division or variable-bit shift or rotate instruction is executing when the interrupt request arrives (see section 2.7 for detailed instruction timing data). As mentioned previously, in a few cases, worst-case latency will span two instructions rather than one.

Internal Interrupts

An INT (interrupt) instruction generates an interrupt immediately upon completion of its execution. The interrupt type coded into the instruction supplies the CPU with the type code needed to call the procedure to process the interrupt. Since any type code may be specified, software interrupts may be used to test interrupt procedures written to service external devices.

*There are a few cases in which an interrupt request is not recognized until after the *following* instruction. Repeat, LOCK and segment override prefixes are considered "part of" the instructions they prefix; no interrupt is recognized between execution of a prefix and an instruction. A MOV (move) to segment register instruction and a POP segment register instruction are treated similarly: no interrupt is recognized until after the following instruction. This mechanism protects a program that is changing to a new stack (by updating SS and SP). If an interrupt were recognized after SS had been changed, but before SP had been altered, the processor would push the flags, CS and IP into the wrong area of memory. It follows from this that whenever a segment register and another value must be updated together, the segment register should be changed first, followed immediately by the instruction that changes the other value. There are also two cases, WAIT and repeated string instructions, where an interrupt request is recognized in the middle of an instruction. In these cases, interrupts are accepted after any completed primitive operation or wait test cycle.

If the overflow flag (OF) is set, an INTO (interrupt on overflow) instruction generates a type 4 interrupt immediately upon completion of its execution.

The CPU itself generates a type 0 interrupt immediately following execution of a DIV or IDIV (divide, integer divide) instruction if the calculated quotient is larger than the specified destination.

If the trap flag (TF) is set, the CPU automatically generates a type 1 interrupt following *every* instruction. This is called single-step execution and is a powerful debugging tool that is discussed in more detail shortly.

All internal interrupts (INT, INTO, divide error, and single-step) share these characteristics:

1. The interrupt type code is either contained in the instruction or is predefined.
2. No INTA bus cycles are run.
3. Internal interrupts cannot be disabled, except for single-step.
4. Any internal interrupt (except single-step) has higher priority than any external interrupt (see table 2-3). If interrupt requests arrive on NMI and/or INTR during execution of an instruction that causes an internal interrupt (e.g., divide error), the internal interrupt is processed first.

Interrupt Pointer Table

The interrupt pointer (or interrupt vector) table (figure 2-30) is the link between an interrupt type code and the procedure that has been designated to service interrupts associated with that code. The interrupt pointer table occupies up to the first 1k bytes of low memory. There may be up to 256 entries in the table, one for each interrupt type

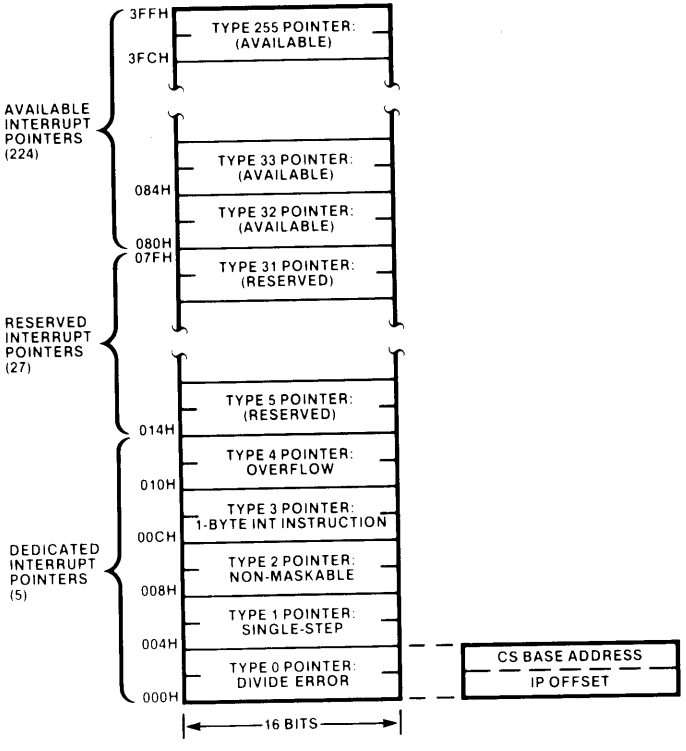


Figure 2-30. Interrupt Pointer Table

that can occur in the system. Each entry in the table is a doubleword pointer containing the address of the procedure that is to service interrupts of that type. The higher-addressed word of the pointer contains the base address of the segment containing the procedure. The lower-addressed word contains the procedure's offset from the beginning of the segment. Since each entry is four bytes long, the CPU can calculate the location of the correct entry for a given interrupt type by simply multiplying (type*4).

Table 2-3. Interrupt Priorities

INTERRUPT	PRIORITY
Divide error, INT n, INTO	highest
NMI	
INTR	
Single-step	lowest

Space at the high end of the table that would be occupied by entries for interrupt types that cannot occur in a given application may be used for other purposes. The dedicated and reserved portions of the interrupt pointer table (locations 0H through 7FH), however, should not be used for any other purpose to insure proper system operation and to preserve compatibility with future Intel hardware and software products.

After pushing the flags onto the stack, the 8086 or 8088 activates an interrupt procedure by executing the equivalent of an intersegment indirect CALL instruction. The target of the "CALL" is the address contained in the interrupt pointer table element located at (type*4). The CPU saves the address of the next instruction by pushing CS and IP onto the stack. These are then replaced by the second and first words of the table element, thus transferring control to the procedure.

If multiple interrupt requests arrive simultaneously, the processor activates the interrupt procedures in priority order. Figure 2-31 shows how procedures would be activated in an extreme case. The processor is running in single-step mode with external interrupts enabled. During execution of a divide instruction, INTR is activated. Furthermore the instruction generates a divide error interrupt. Figure 2-31 shows that the interrupts are recognized in turn, in the order of their priorities except for INTR. INTR is not recognized until after the following instruction because recognition of the earlier interrupts cleared IF. Of couse interrupts could be reenabled in any of the interrupt response routines if earlier response to INTR is desired.

As figure 2-31 shows, all main-line code is executed in single-step mode. Also, because of the order of interrupt processing, the opportunity exists in each occurrence of the single-step routine to select whether pending interrupt routines (divide error and INTR routines in this example) are executed at full speed or in single-step mode.

Interrupt Procedures

When an interrupt service procedure is entered, the flags, CS, and IP are pushed onto the stack and TF and IF are cleared. The procedure may reenable external interrupts with the STI (set interrupt-enable flag) instruction, thus allowing itself to be interrupted by a request on INTR. (Note, however, that interrupts are not actually enabled until the instruction *following* STI has executed.) An interrupt procedure always may be interrupted by a request arriving on NMI. Software- or processor-initiated interrupts occurring within the procedure also will interrupt the procedure. Care must be taken in interrupt procedures that the type of interrupt being serviced by the procedure does not itself inadvertently occur within the procedure. For example, an attempt to divide by 0 in the divide error (type 0) interrupt procedure may result in the procedure being reentered endlessly. Enough stack space must be available to accommodate the maximum depth of interrupt nesting that can occur in the system.

Like all procedures, interrupt procedures should save any registers they use before updating them, and restore them before terminating. It is good practice for an interrupt procedure to enable external interrupts for all but "critical sections" of code (those sections that cannot be interrupted without risking erroneous results). If external interrupts are disabled for too long in a procedure, interrupt requests on INTR can potentially be lost.

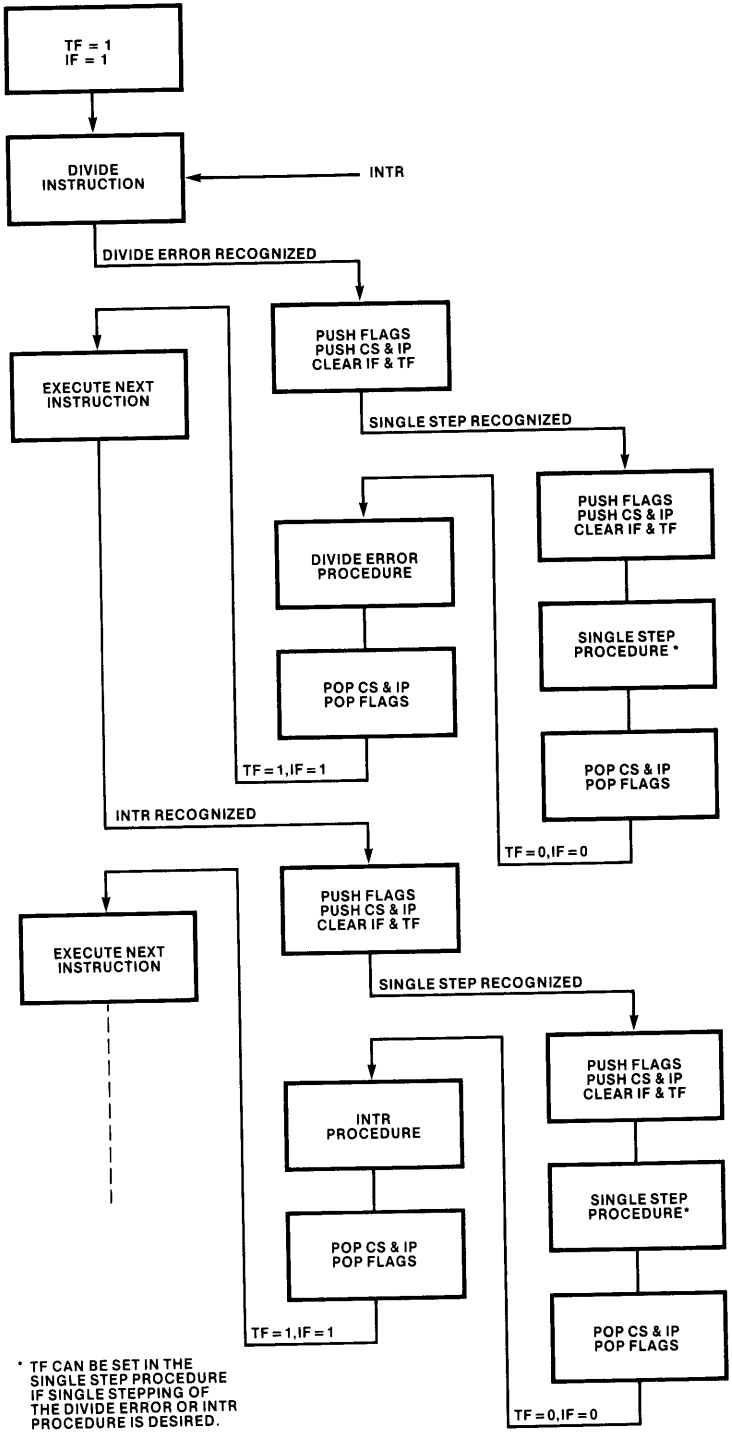


Figure 2-31. Processing Simultaneous Interrupts

All interrupt procedures should be terminated with an IRET (interrupt return) instruction. The IRET instruction assumes that the stack is in the same condition as it was when the procedure was entered. It pops the top three stack words into IP, CS and the flags, thus returning to the instruction that was about to be executed when the interrupt procedure was activated.

The actual processing done by the procedure is dependent upon the application. If the procedure is servicing an external device, it should output a command to the device instructing it to remove its interrupt request. It might then read status information from the device, determine the cause of the interrupt and then take action accordingly. Section 2.10 contains three typical interrupt procedure examples.

Software-initiated interrupt procedures may be used as service routines ("supervisor calls") for other programs in the system. In this case, the interrupt procedure is activated when a program, rather than an external device, needs attention. (The "attention" might be to search a file for a record, send a message to another program, request an allocation of free memory, etc.) Software interrupt procedures can be advantageous in systems that dynamically relocate programs during execution. Since the interrupt pointer table is at a fixed storage location, procedures may "call" each other through the table by issuing software interrupt instructions. This provides a stable communication "exchange" that is independent of procedure addresses. The interrupt procedures may themselves be moved so long as the interrupt pointer table always is updated to provide the linkage from the "calling" program via the interrupt type code.

Single-Step (Trap) Interrupt

When TF (the trap flag) is set, the 8086 or 8088 is said to be in single-step mode. In this mode, the processor automatically generates a type 1 interrupt after each instruction. Recall that as part of its interrupt processing, the CPU automatically pushes the flags onto the stack and then clears TF and IF. Thus the processor is *not* in single-step mode when the single-step interrupt procedure is entered; it runs normally. When the single-step procedure terminates, the old flag image is restored from the stack, placing the CPU back into single-step mode.

Single-stepping is a valuable debugging tool. It allows the single-step procedure to act as a "window" into the system through which operation can be observed instruction-by-instruction. A single-step interrupt procedure, for example, can print or display register contents, the value of the instruction pointer (it is on the stack), key memory variables, etc., as they change after each instruction. In this way the exact flow of a program can be traced in detail, and the point at which discrepancies occur can be determined. Other possible services that could be provided by a single-step routine include:

- Writing a message when a specified memory location or I/O port changes value (or equals a specified value).
- Providing diagnostics selectively (only for certain instruction addresses for instance).
- Letting a routine execute a number of times before providing diagnostics.

The 8086 and 8088 do not have instructions for setting or clearing TF directly. Rather, TF can be changed by modifying the flag-image on the stack. The PUSHF and POPF instructions are available for pushing and popping the flags directly (TF can be set by ORing the flag-image with 0100H and cleared by ANDing it with FEFFH). After TF is set in this manner, the first single-step interrupt occurs after the first instruction following the IRET from the single-step procedure.

If the processor is single-stepping, it processes an interrupt (either internal or external) as follows. Control is passed normally (flags, CS and IP are pushed) to the procedure designated to handle the type of interrupt that has occurred. However, before the first instruction of that procedure is executed, the single-step interrupt is "recognized" and control is passed normally (flags, CS and IP are pushed) to the type 1 interrupt procedure. When single-step procedure terminates, control returns to the previous interrupt procedure. Figure 2-31 illustrates this process in a case where two interrupts occur when the processor is in single-step mode.

Breakpoint Interrupt

A type 3 interrupt is dedicated to the breakpoint interrupt. A breakpoint is generally any place in a program where normal execution is arrested so

that some sort of special processing may be performed. Breakpoints typically are inserted into programs during debugging as a way of displaying registers, memory locations, etc., at crucial points in the program.

The INT 3 (breakpoint) instruction is one byte long. This makes it easy to "plant" a breakpoint anywhere in a program. Section 2.10 contains an example that shows how a breakpoint may be set and how a breakpoint procedure may be used to place the processor into single-step mode.

The breakpoint instruction also may be used to "patch" a program (insert new instructions) without recompiling or reassembling it. This may be done by saving an instruction byte, and replacing it with an INT 3 (CCH) machine instruction. The breakpoint procedure would contain the new machine instructions, plus code to restore the saved instruction byte and decrement IP on the stack before returning, so that the displaced instruction would be executed after the patch instructions. The breakpoint example in section 2.10 illustrates these principles.

Note that patching a program requires machine-instruction programming and should be undertaken with considerable caution; it is easy to add new bugs to a program in an attempt to correct existing ones. Note also that a patch is only a temporary measure to be used in exceptional conditions. The affected code should be updated and retranslated as soon as possible.

System Reset

The 8086/8088 RESET line provides an orderly way to start or restart an executing system. When the processor detects the positive-going edge of a pulse on RESET, it terminates all activities until the signal goes low, at which time it initializes the system as shown in table 2-4.

Since the code segment register contains FFFFH and the instruction pointer contains 0H, the processor executes its first instruction following system reset from absolute memory location FFFF0H. This location normally contains an intersegment direct JMP instruction whose target is the actual beginning of the system program. The LOC-86 utility supplies this JMP instruction from information in the program that identifies its first instruction. As external (maskable) interrupts are disabled by system reset, the system software should reenable interrupts as soon as the system is initialized to the point where they can be processed.

Table 2-4. CPU State Following RESET

CPU COMPONENT	CONTENT
Flags	Clear
Instruction Pointer	0000H
CS Register	FFFFH
DS Register	0000H
SS Register	0000H
ES Register	0000H
Queue	Empty

Instruction Queue Status

When configured in maximum mode, the 8086 and 8088 provide information about instruction queue operations on lines QS0 and QS1. Table 2-5 interprets the four states that these lines can represent.

The queue status lines are provided for external processors that receive instructions and/or operands via the 8086/8088 ESC (escape) instruction (see sections 2.5 and 2.8). Such a processor may monitor the bus to see when an ESC instruction is fetched and then track the instruction through the queue to determine when (and if) the instruction is executed.

Table 2-5. Queue Status Signals (Maximum Mode Only)

QS_0	QS_1	QUEUE OPERATION IN LAST CLK CYCLE
0	0	No operation; default value
0	1	First byte of an instruction was taken from the queue
1	0	Queue was reinitialized
1	1	Subsequent byte of an instruction was taken from the queue

Processor Halt

When the HLT (halt) instruction (see section 2.7) is executed, the 8086 or 8088 enters the halt state. This condition may be interpreted as "stop all

operations until an external interrupt occurs or the system is reset." No signals are floated during the halt state, and the content of the address and data buses is undefined. A bus hold request arriving on the HOLD line (minimum mode) or either request/grant line (maximum mode) is acknowledged normally while the processor is halted.

The halt state can be used when an event prevents the system from functioning correctly. An example might be a power-fail interrupt. After recognizing that loss of power is imminent, the CPU could use the remaining time to move registers, flags and vital variables to (for example) a battery-powered CMOS RAM area and then halt until the return of power was signaled by an interrupt or system reset.

Status Lines

When configured in maximum mode, the 8086 and 8088 emit eight status signals that can be used by external devices. Lines $\overline{S0}$, $\overline{S1}$ and $\overline{S2}$ identify the type of bus cycle that the CPU is starting to execute (table 2-6). These lines are typically decoded by the 8288 Bus Controller. S3 and S4 indicate which segment register was used to construct the physical address being used in this bus cycle (see table 2-7). Line S5 reflects the state of the interrupt-enable flag. S6 is always 0. S7 is a spare line whose content is undefined.

Table 2-6. Bus Cycle Status Signals

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	TYPES OF BUS CYCLE
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O
0	1	0	Write I/O
0	1	1	HALT
1	0	0	Instruction Fetch
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive; no bus cycle

Table 2-7. Segment Register Status Lines

S_4	S_3	SEGMENT REGISTER
0	0	ES
0	1	SS
1	0	CS or none (I/O or Interrupt Vector)
1	1	DS

2.7 Instruction Set

The 8086 and 8088 execute exactly the same instructions. This instruction set includes equivalents to the instructions typically found in previous microprocessors, such as the 8080/8085. Significant new operations include:

- multiplication and division of signed and unsigned binary numbers as well as unpacked decimal numbers,
- move, scan and compare operations for strings up to 64k bytes in length,
- non-destructive bit testing,
- byte translation from one code to another,
- software-generated interrupts, and
- a group of instructions that can help coordinate the activities of multiprocessor systems.

These instructions treat different types of operands uniformly. Nearly every instruction can operate on either byte or word data. Register, memory and immediate operands may be specified interchangeably in most instructions (except, of course, that immediate values may only serve as "source" and not "destination" operands). In particular, memory variables can be added to, subtracted from, shifted, compared, and so on, in place, without moving them in and out of registers. This saves instructions, registers, and execution time in assembly language programs. In high-level languages, where most variables are memory based, compilers, such as PL/M-86, can produce faster and shorter object programs.

The 8086/8088 instruction set can be viewed as existing at two levels: the assembly level and the machine level. To the assembly language programmer, the 8086 and 8088 appear to have a repertoire of about 100 instructions. One MOV (move) instruction, for example, transfers a byte or a word from a register or a memory location or an immediate value to either a register or a memory location. The 8086 and 8088 CPUs, however, recognize 28 different MOV machine instructions ("move byte register to memory," "move word immediate to register," etc.). The ASM-86 assembler translates the assembly-level instructions written by a programmer into the

machine-level instructions that are actually executed by the 8086 or 8088. Compilers such as PL/M-86 translate high-level language statements directly into machine-level instructions.

The two levels of the instruction set address two different requirements: efficiency and simplicity. The numerous—there are about 300 in all—forms of machine-level instructions allow these instructions to make very efficient use of storage. For example, the machine instruction that increments a memory operand is three or four bytes long because the address of the operand must be encoded in the instruction. To increment a register, however, does not require as much information, so the instruction can be shorter. In fact, the 8086 and 8088 have eight different machine-level instructions that increment a different 16-bit register; these instructions are only one byte long.

If a programmer had to write one instruction to increment a register, another to increment a memory variable, etc., the benefit of compact instructions would be offset by the difficulty of programming. The assembly-level instructions simplify the programmer's view of the instruction set. The programmer writes one form of the INC (increment) instruction and the ASM-86 assembler examines the operand to determine which machine-level instruction to generate.

This section presents the 8086/8088 instruction set from two perspectives. First, the assembly-level instructions are described in functional terms. The assembly-level instructions are then presented in a reference table that breaks out all permissible operand combinations with execution times and machine instruction length, plus the effect that the instruction has on the CPU flags. Machine-level instruction encoding and decoding are covered in section 4.2.

Data Transfer Instructions

The 14 data transfer instructions (table 2-8) move single bytes and words between memory and registers as well as between register AL or AX and I/O ports. The stack manipulation instructions are included in this group as are instructions for transferring flag contents and for loading segment registers.

Table 2-8. Data Transfer Instructions

GENERAL PURPOSE	
MOV	Move byte or word
PUSH	Push word onto stack
POP	Pop word off stack
XCHG	Exchange byte or word
XLAT	Translate byte
INPUT/OUTPUT	
IN	Input byte or word
OUT	Output byte or word
ADDRESS OBJECT	
LEA	Load effective address
LDS	Load pointer using DS
LES	Load pointer using ES
FLAG TRANSFER	
LAHF	Load AH register from flags
SAHF	Store AH register in flags
PUSHF	Push flags onto stack
POPF	Pop flags off stack

General Purpose Data Transfers

MOV *destination,source*

MOV transfers a byte or a word from the source operand to the destination operand.

PUSH *source*

PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed to by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.

POP *destination*

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand, and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

XCHG *destination,source*

XCHG (exchange) switches the contents of the source and destination (byte or word) operands. When used in conjunction with the LOCK prefix, XCHG can test and set a semaphore that controls access to a resource shared by multiple processors (see section 2.5).

XLAT *translate-table*

XLAT (translate) replaces a byte in the AL register with a byte from a 256-byte, user-coded translation table. Register BX is assumed to point to the beginning of the table. The byte in AL is used as an index into the table and is replaced by the byte at the offset in the table corresponding to AL's binary value. The first byte in the table has an offset of 0. For example, if AL contains 5H, and the sixth element of the translation table contains 33H, then AL will contain 33H following the instruction. XLAT is useful for translating characters from one code to another, the classic example being ASCII to EBCDIC or the reverse.

IN *accumulator,port*

IN transfers a byte or a word from an input port to the AL register or the AX register, respectively. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in the DX register, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.

OUT *port,accumulator*

OUT transfers a byte or a word from the AL register or the AX register, respectively, to an output port. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in register DX, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.

Address Object Transfers

These instructions manipulate the *addresses* of variables rather than the contents or values of variables. They are most useful for list processing, based variables, and string operations.

LEA *destination,source*

LEA (load effective address) transfers the offset of the source operand (rather than its value) to the destination operand. The source operand must be a memory operand, and the destination operand must be a 16-bit general register. LEA does not affect any flags. The XLAT and string instructions assume that certain registers point to operands; LEA can be used to load these registers (e.g., loading BX with the address of the translate table used by the XLAT instruction).

LDS *destination,source*

LDS (load pointer using DS) transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register DS. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register DS. Specifying SI as the destination operand is a convenient way to prepare to process a source string that is not in the current data segment (string instructions assume that the source string is located in the current data segment and that SI contains the offset of the string).

LES *destination,source*

LES (load pointer using ES) transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register ES. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register ES. Specifying DI as the destination operand is a convenient way to prepare to process a destination string that is not in the current extra segment. (The destination string must be located in the extra segment, and DI must contain the offset of the string.)

Flag Transfers

LAHF

LAHF (load register AH from flags) copies SF, ZF, AF, PF and CF (the 8080/8085 flags) into bits 7, 6, 4, 2 and 0, respectively, of register AH

(see figure 2-32). The content of bits 5, 3 and 1 is undefined; the flags themselves are not affected. LAHF is provided primarily for converting 8080/8085 assembly language programs to run on an 8086 or 8088.

SAHF

SAHF (store register AH into flags) transfers bits 7, 6, 4, 2 and 0 from register AH into SF, ZF, AF, PF and CF, respectively, replacing whatever values these flags previously had. OF, DF, IF and TF are not affected. This instruction is provided for 8080/8085 compatibility.

PUSHF

PUSHF decrements SP (the stack pointer) by two and then transfers all flags to the word at the top of stack pointed to by SP (see figure 2-32). The flags themselves are not affected.

POPF

POPF transfers specific bits from the word at the current top of stack (pointed to by register SP) into the 8086/8088 flags, replacing whatever values the flags previously contained (see figure 2-32). SP is then incremented by two to point to the new top of stack. PUSHF and POPF allow a procedure to save and restore a calling program's flags. They also allow a program to change the setting of TF (there is no instruction for updating this flag directly). The change is accomplished by pushing the flags, altering bit 8 of the memory-image and then popping the flags.

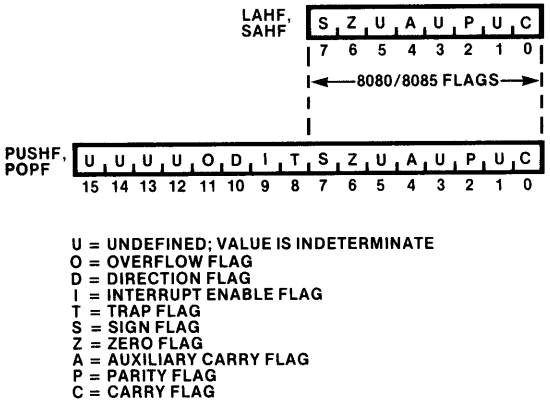


Figure 2-32. Flag Storage Formats

Arithmetic Instructions

Arithmetic Data Formats

8086 and 8088 arithmetic operations (table 2-9) may be performed on four types of numbers: unsigned binary, signed binary (integers), unsigned packed decimal and unsigned unpacked decimal (see table 2-10). Binary numbers may be 8 or 16 bits long. Decimal numbers are stored in bytes, 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 that represent valid numbers for the type of instruction being performed. Invalid data may produce unpredictable results.

Table 2-9. Arithmetic Instructions

ADDITION	
ADD	Add byte or word
ADC	Add byte or word with carry
INC	Increment byte or word by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
SUBTRACTION	
SUB	Subtract byte or word
SBB	Subtract byte or word with borrow
DEC	Decrement byte or word by 1
NEG	Negate byte or word
CMP	Compare byte or word
AAS	ASCII adjust for subtraction
DAS	Decimal adjust for subtraction
MULTIPLICATION	
MUL	Multiply byte or word unsigned
IMUL	Integer multiply byte or word
AAM	ASCII adjust for multiply
DIVISION	
DIV	Divide byte or word unsigned
IDIV	Integer divide byte or word
AAD	ASCII adjust for division
CBW	Convert byte to word
CWD	Convert word to doubleword

Table 2-10. Arithmetic Interpretation of 8-Bit Numbers

HEX	BIT PATTERN	UNSIGNED BINARY	SIGNED BINARY	UNPACKED DECIMAL	PACKED DECIMAL
07	0 0 0 0 0 1 1 1	7	+7	7	7
89	1 0 0 0 1 0 0 1	137	−119	invalid	89
C5	1 1 0 0 0 1 0 1	197	−59	invalid	invalid

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

Signed binary numbers (integers) may be either 8 or 16 bits long. The high-order (leftmost) bit is interpreted as the number's sign: 0 = positive and 1 = negative. Negative numbers are represented in standard two's complement notation. Since the high-order bit is used for a sign, the range of an 8-bit integer is −128 through +127; 16-bit integers may range from −32,768 through +32,767. The value zero has a positive sign. Multiplication and division operations are provided for signed binary numbers. Addition and subtraction are performed with the unsigned binary instructions. Conditional jump instructions, as well as an "interrupt on overflow" instruction, can be used following an unsigned operation on an integer to detect overflow into the sign bit.

Packed decimal numbers are stored as unsigned byte quantities. The byte is treated as having one decimal digit in each half-byte (nibble); the digit in the high-order half-byte is the most significant. Hexadecimal values 0-9 are valid in each half-byte, and the range of a packed decimal number is 0-99. Addition and subtraction are performed in two steps. First an unsigned binary instruction is used to produce an intermediate result in register AL. Then an adjustment operation is performed which changes the intermediate value in AL to a final correct packed decimal result. Multiplication and division adjustments are not available for packed decimal numbers.

Unpacked decimal numbers are stored as unsigned byte quantities. The magnitude of the number is determined from the low-order half-byte; hexadecimal values 0-9 are valid and are interpreted as decimal numbers. The high-order half-byte must be zero for multiplication and division; it may contain any value for addition and subtraction. Arithmetic on unpacked decimal numbers is performed in two steps. The unsigned binary addition, subtraction and multiplication operations are used to produce an intermediate result in register AL. An adjustment instruction then changes the value in AL to a final correct unpacked decimal number. Division is performed similarly, except that the adjustment is carried out on the numerator operand in register AL first, then a following unsigned binary division instruction produces a correct result.

Unpacked decimal numbers are similar to the ASCII character representations of the digits 0-9. Note, however, that the high-order half-byte of an ASCII numeral is always 3H. Unpacked decimal arithmetic may be performed on ASCII numeric characters under the following conditions:

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

Arithmetic Instructions and Flags

The 8086/8088 arithmetic instructions post certain characteristics of the result of the operation to six flags. Most of these flags can be tested by following the arithmetic instruction with a conditional jump instruction; the INTO (interrupt on overflow) instruction also may be used. The

various instructions affect the flags differently, as explained in the instruction descriptions. However, they follow these general rules:

- CF (carry flag): If an addition results in a carry out of the high-order bit of the result, then CF is set; otherwise CF is cleared. If a subtraction results in a borrow into the high-order bit of the result, then CF is set; otherwise CF is cleared. Note that a *signed* carry is indicated by CF ≠ OF. CF can be used to detect an unsigned overflow. Two instructions, ADC (add with carry) and SBB (subtract with borrow), incorporate the carry flag in their operations and can be used to perform multibyte (e.g., 32-bit, 64-bit) addition and subtraction.

- AF (auxiliary carry flag): If an addition results in a carry out of the low-order half-byte of the result, then AF is set; otherwise AF is cleared. If a subtraction results in a borrow into the low-order half-byte of the result, then AF is set; otherwise AF is cleared. The auxiliary carry flag is provided for the decimal adjust instructions and ordinarily is not used for any other purpose.

- SF (sign flag): Arithmetic and logical instructions set the sign flag equal to the high-order bit (bit 7 or 15) of the result. For signed binary numbers, the sign flag will be 0 for positive results and 1 for negative results (so long as overflow does not occur). A conditional jump instruction can be used following addition or subtraction to alter the flow of the program depending on the sign of the result. Programs performing unsigned operations typically ignore SF since the high-order bit of the result is interpreted as a digit rather than a sign.

- ZF (zero flag): If the result of an arithmetic or logical operation is zero, then ZF is set; otherwise ZF is cleared. A conditional jump instruction can be used to alter the flow of the program if the result is or is not zero.

- PF (parity flag): If the low-order eight bits of an arithmetic or logical result contain an even number of 1-bits, then the parity flag is set; otherwise it is cleared. PF is provided for 8080/8085 compatibility; it also can be used to check ASCII characters for correct parity.

- OF (overflow flag): If the result of an operation is too large a positive number, or too small a negative number to fit in the destination operand (excluding the sign bit), then OF is set; otherwise OF is cleared. OF thus indicates signed arithmetic overflow; it can be tested with a conditional jump or the INTO (interrupt on overflow) instruction. OF may be ignored when performing unsigned arithmetic.

Addition

ADD *destination,source*

The sum of the two operands, which may be bytes or words, replaces the destination operand. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADD updates AF, CF, OF, PF, SF and ZF.

ADC *destination,source*

ADC (Add with Carry) sums the operands, which may be bytes or words, adds one if CF is set and replaces the destination operand with the result. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADC updates AF, CF, OF, PF, SF and ZF. Since ADC incorporates a carry from a previous operation, it can be used to write routines to add numbers longer than 16 bits.

INC *destination*

INC (Increment) adds one to the destination operand. The operand may be a byte or a word and is treated as an unsigned binary number (see AAA and DAA). INC updates AF, OF, PF, SF and ZF; it does not affect CF.

AAA

AAA (ASCII Adjust for Addition) changes the contents of register AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAA updates AF and CF; the content of OF, PF, SF and ZF is undefined following execution of AAA.

DAA

DAA (Decimal Adjust for Addition) corrects the result of previously adding two valid packed decimal operands (the destination operand must have been register AL). DAA changes the content of AL to a pair of valid packed decimal digits. It updates AF, CF, PF, SF and ZF; the content of OF is undefined following execution of DAA.

Subtraction

SUB *destination,source*

The source operand is subtracted from the destination operand, and the result replaces the destination operand. The operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SUB updates AF, CF, OF, PF, SF and ZF.

SBB *destination,source*

SBB (Subtract with Borrow) subtracts the source from the destination, subtracts one if CF is set, and returns the result to the destination operand. Both operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SBB updates AF, CF, OF, PF, SF and ZF. Since it incorporates a borrow from a previous operation, SBB may be used to write routines that subtract numbers longer than 16 bits.

DEC *destination*

DEC (Decrement) subtracts one from the destination, which may be a byte or a word. DEC updates AF, OF, PF, SF, and ZF; it does not affect CF.

NEG *destination*

NEG (Negate) subtracts the destination operand, which may be a byte or a word, from 0 and returns the result to the destination. This forms the two's complement of the number, effectively reversing the sign of an integer. If the operand is zero, its sign is not changed. Attempting to negate a byte containing −128 or a word containing −32,768 causes no change to the operand and sets OF. NEG updates AF, CF, OF, PF, SF and ZF. CF is always set except when the operand is zero, in which case it is cleared.

CMP *destination,source*

CMP (Compare) subtracts the source from the destination, which may be bytes or words, but does not return the result. The operands are unchanged, but the flags are updated and can be tested by a subsequent conditional jump instruction. CMP updates AF, CF, OF, PF, SF and ZF. The comparison reflected in the flags is that of the destination to the source. If a CMP instruction is followed by a JG (jump if greater) instruction, for example, the jump is taken if the destination operand is greater than the source operand.

AAS

AAS (ASCII Adjust for Subtraction) corrects the result of a previous subtraction of two valid unpacked decimal operands (the destination operand must have been specified as register AL). AAS changes the content of AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAS updates AF and CF; the content of OF, PF, SF and ZF is undefined following execution of AAS.

DAS

DAS (Decimal Adjust for Subtraction) corrects the result of a previous subtraction of two valid packed decimal operands (the destination operand must have been specified as register AL). DAS changes the content of AL to a pair of valid packed decimal digits. DAS updates AF, CF, PF, SF and ZF; the content of OF is undefined following execution of DAS.

Multiplication

MUL *source*

MUL (Multiply) performs an unsigned multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length

result is returned in AH and AL. If the source operand is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. The operands are treated as unsigned binary numbers (see AAM). If the upper half of the result (AH for byte source, DX for word source) is nonzero, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF and ZF is undefined following execution of MUL.

IMUL *source*

IMUL (Integer Multiply) performs a signed multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length result is returned in AH and AL. If the source is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. If the upper half of the result (AH for byte source, DX for word source) is not the sign extension of the lower half of the result, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF and ZF is undefined following execution of IMUL.

AAM

AAM (ASCII Adjust for Multiply) corrects the result of a previous multiplication of two valid unpacked decimal operands. A valid 2-digit unpacked decimal number is derived from the content of AH and AL and is returned to AH and AL. The high-order half-bytes of the multiplied operands must have been 0H for AAM to produce a correct result. AAM updates PF, SF and ZF; the content of AF, CF and OF is undefined following execution of AAM.

Division

DIV *source*

DIV (divide) performs an unsigned division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH. The single-length quotient is returned in AL, and the single-length remainder is returned in AH. If the source operand is a word, it is divided into the double-length dividend in registers AX and DX. The single-length quotient is returned in AX, and the single-length remainder is returned in DX. If the quotient exceeds the capacity of its destination register (FFH for byte source, FFFFFH for word source), as when division by zero is attempted, a type 0 interrupt is generated, and the quotient and remainder are undefined. Nonintegral quotients are truncated to integers. The content of AF, CF, OF, PF, SF and ZF is undefined following execution of DIV.

IDIV *source*

IDIV (Integer Divide) performs a signed division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH; the single-length quotient is returned in AL, and the single-length remainder is returned in AH. For byte integer division, the maximum positive quotient is +127 (7FH) and the minimum negative quotient is −127 (81H). If the source operand is a word, it is divided into the double-length dividend in registers AX and DX; the single-length quotient is returned in AX, and the single-length remainder is returned in DX. For word integer division, the maximum positive quotient is +32,767 (7FFFH) and the minimum negative quotient is −32,767 (8001H). If the quotient is positive and exceeds the maximum, or is negative and is less than the minimum, the quotient and remainder are undefined, and a type 0 interrupt is generated. In particular, this occurs if division by 0 is attempted. Nonintegral quotients are truncated (toward 0) to integers, and the remainder has the same sign as the dividend. The content of AF, CF, OF, PF, SF and ZF is undefined following IDIV.

AAD

AAD (ASCII Adjust for Division) modifies the numerator in AL *before* dividing two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. AH must be zero for the subse-

quent DIV to produce the correct result. The quotient is returned in AL, and the remainder is returned in AH; both high-order half-bytes are zeroed. AAD updates PF, SF and ZF; the content of AF, CF and OF is undefined following execution of AAD.

CBW

CBW (Convert Byte to Word) extends the sign of the byte in register AL throughout register AH. CBW does not affect any flags. CBW can be used to produce a double-length (word) dividend from a byte prior to performing byte division.

CWD

CWD (Convert Word to Doubleword) extends the sign of the word in register AX throughout register DX. CWD does not affect any flags. CWD can be used to produce a double-length (doubleword) dividend from a word prior to performing word division.

Bit Manipulation Instructions

The 8086 and 8088 provide three groups of instructions (table 2-11) for manipulating bits within both bytes and words: logical, shifts and rotates.

Table 2-11. Bit Manipulation Instructions

LOGICALS	
NOT	"Not" byte or word
AND	"And" byte or word
OR	"Inclusive or" byte or word
XOR	"Exclusive or" byte or word
TEST	"Test" byte or word
SHIFTS	
SHL/SAL	Shift logical/arithmetic left byte or word
SHR	Shift logical right byte or word
SAR	Shift arithmetic right byte or word
ROTATES	
ROL	Rotate left byte or word
ROR	Rotate right byte or word
RCL	Rotate through carry left byte or word
RCR	Rotate through carry right byte or word

Logical

The logical instructions include the boolean operators "not," "and," "inclusive or," and "exclusive or," plus a TEST instruction that sets the flags, but does not alter either of its operands.

AND, OR, XOR and TEST affect the flags as follows: The overflow (OF) and carry (CF) flags are always cleared by logical instructions, and the content of the auxiliary carry (AF) flag is always undefined following execution of a logical instruction. The sign (SF), zero (ZF) and parity (PF) flags are always posted to reflect the result of the operation and can be tested by conditional jump instructions. The interpretation of these flags is the same as for arithmetic instructions. SF is set if the result is negative (high-order bit is 1), and is cleared if the result is positive (high-order bit is 0). ZF is set if the result is zero, cleared otherwise. PF is set if the result contains an even number of 1-bits (has even parity) and is cleared if the number of 1-bits is odd (the result has odd parity). Note that NOT has no effect on the flags.

NOT *destination*

NOT inverts the bits (forms the one's complement) of the byte or word operand.

AND *destination,source*

AND performs the logical "and" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if both corresponding bits of the original operands are set; otherwise the bit is cleared.

OR *destination,source*

OR performs the logical "inclusive or" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if either or both corresponding bits in the original operands are set; otherwise the result bit is cleared.

XOR *destination,source*

XOR (Exclusive Or) performs the logical "exclusive or" of the two operands and returns the result to the destination operand. A bit in the

result is set if the corresponding bits of the original operands contain opposite values (one is set, the other is cleared); otherwise the result bit is cleared.

TEST *destination,source*

TEST performs the logical "and" of the two operands (byte or word), updates the flags, but does not return the result, i.e., neither operand is changed. If a TEST instruction is followed by a JNZ (jump if not zero) instruction, the jump will be taken if there are any corresponding 1-bits in both operands.

Shifts

The bits in bytes and words may be shifted arithmetically or logically. Up to 255 shifts may be performed, according to the value of the count operand coded in the instruction. The count may be specified as the constant 1, or as register CL, allowing the shift count to be a variable supplied at execution time. Arithmetic shifts may be used to multiply and divide binary numbers by powers of two (see note in description of SAR). Logical shifts can be used to isolate bits in bytes or words.

Shift instructions affect the flags as follows. AF is always undefined following a shift operation. PF, SF and ZF are updated normally, as in the logical instructions. CF always contains the value of the last bit shifted out of the destination operand. The content of OF is always undefined following a multibit shift. In a single-bit shift, OF is set if the value of the high-order (sign) bit was changed by the operation; if the sign bit retains its original value, OF is cleared.

SHL/SAL *destination,count*

SHL and SAL (Shift Logical Left and Shift Arithmetic Left) perform the same operation and are physically the same instruction. The destination byte or word is shifted left by the number of bits specified in the count operand. Zeros are shifted in on the right. If the sign bit retains its original value, then OF is cleared.

SHR *destination,source*

SHR (Shift Logical Right) shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Zeros are shifted in on the left. If the sign bit retains its original value, then OF is cleared.

SAR *destination,count*

SAR (Shift Arithmetic Right) shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Bits equal to the original high-order (sign) bit are shifted in on the left, preserving the sign of the original value. Note that SAR does not produce the same result as the dividend of an "equivalent" IDIV instruction if the destination operand is negative and 1-bits are shifted out. For example, shifting −5 right by one bit yields −3, while integer division of −5 by 2 yields −2. The difference in the instructions is that IDIV truncates all numbers toward zero, while SAR truncates positive numbers toward zero and negative numbers toward negative infinity.

Rotates

Bits in bytes and words also may be rotated. Bits rotated out of an operand are not lost as in a shift, but are "circled" back into the other "end" of the operand. As in the shift instructions, the number of bits to be rotated is taken from the count operand, which may specify either a constant of 1, or the CL register. The carry flag may act as an extension of the operand in two of the rotate instructions, allowing a bit to be isolated in CF and then tested by a JC (jump if carry) or JNC (jump if not carry) instruction.

Rotates affect only the carry and overflow flags. CF always contains the value of the last bit rotated out. On multibit rotates, the value of OF is always undefined. In single-bit rotates, OF is set if the operation changes the high-order (sign) bit of the destination operand. If the sign bit retains its original value, OF is cleared.

ROL *destination,count*

ROL (Rotate Left) rotates the destination byte or word left by the number of bits specified in the count operand.

ROR *destination,count*

ROR (Rotate Right) operates similar to ROL except that the bits in the destination byte or word are rotated right instead of left.

RCL *destination,count*

RCL (Rotate through Carry Left) rotates the bits in the byte or word destination operand to the left by the number of bits specified in the count operand. The carry flag (CF) is treated as "part of" the destination operand; that is, its value is rotated into the low-order bit of the destination, and itself is replaced by the high-order bit of the destination.

RCR *destination,count*

RCR (Rotate through Carry Right) operates exactly like RCL except that the bits are rotated right instead of left.

String Instructions

Five basic string operations, called primitives, allow strings of bytes or words to be operated on, one element (byte or word) at a time. Strings of up to 64k bytes may be manipulated with these instructions. Instructions are available to move, compare and scan for a value, as well as for moving string elements to and from the accumulator (see table 2-12). These basic operations may be preceded by a special one-byte prefix that causes the instruction to be repeated by the hardware, allowing long strings to be processed much faster than would be possible with a software loop. The repetitions can be terminated by a variety of conditions, and a repeated operation may be interrupted and resumed.

The string instructions operate quite similarly in many respects; the common characteristics are covered here and in table 2-13 and figure 2-33 rather than in the descriptions of the individual instructions. A string instruction may have a source operand, a destination operand, or both. The hardware assumes that a source string resides in the current data segment; a segment prefix byte may be used to override this assumption. A destination string must be in the current extra segment. The assembler checks the attributes of the operands to determine if the elements of the strings are bytes or words. The assembler does not, however, use the operand names to address the strings. Rather, the content of register SI (source index) is used as an offset to address the current element of the source string, and the content of register DI (destination index) is taken as the offset of the current destination string element. These registers must be initialized to point to the source/destination strings before executing the string instruction; the LDS, LES and LEA instructions are useful in this regard.

Table 2-12. String Instructions

REP	Repeat
REPE/REPZ	Repeat while equal/zero
REPNE/REPNZ	Repeat while not equal/not zero
MOVS	Move byte or word string
MOVSB/MOVSW	Move byte or word string
CMPS	Compare byte or word string
SCAS	Scan byte or word string
LODS	Load byte or word string
STOS	Store byte or word string

Table 2-13. String Instruction Register and Flag Use

SI	Index (offset) for source string
DI	Index (offset) for destination string
CX	Repetition counter
AL/AX	Scan value Destination for LODS Source for STOS
DF	0 = auto-increment SI, DI 1 = auto-decrement SI, DI
ZF	Scan/compare terminator

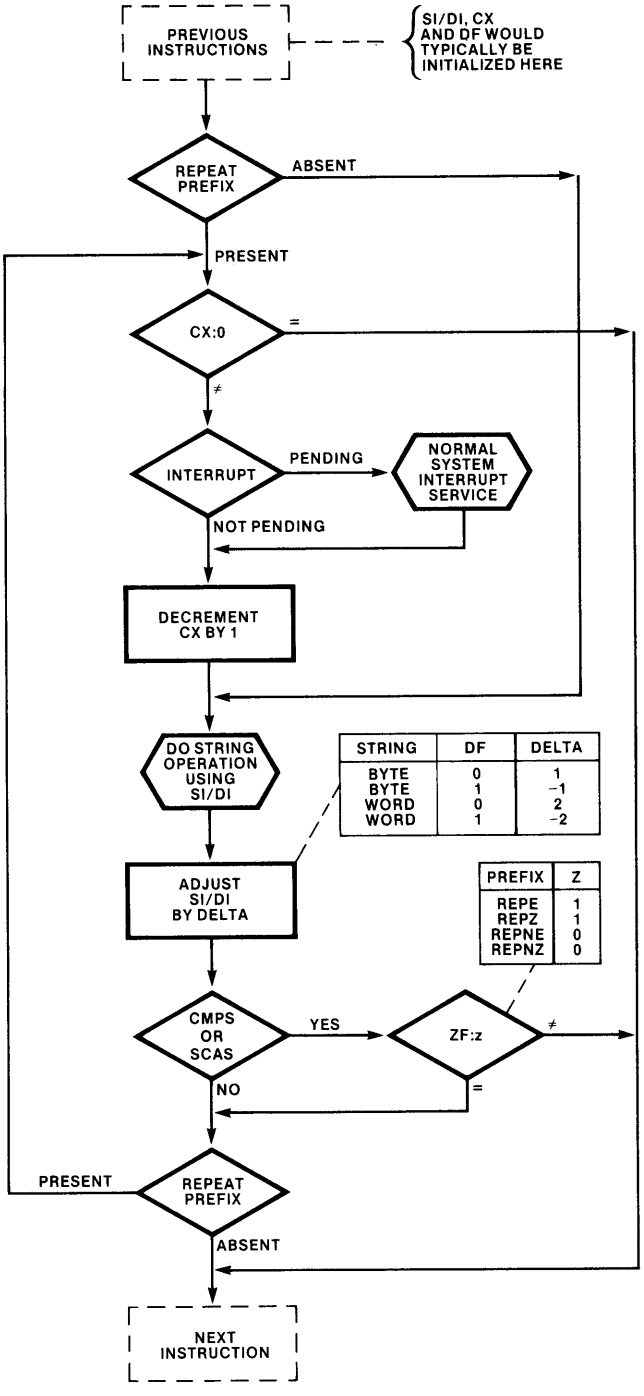


Figure 2-33. String Operation Flow

The string instructions automatically update SI and/or DI in anticipation of processing the next string element. The setting of DF (the direction flag) determines whether the index registers are auto-incremented (DF = 0) or auto-decremented (DF = 1). If byte strings are being processed, SI and/or DI is adjusted by1; the adjustment is 2 for word strings.

If a Repeat prefix has been coded, then register CX (count register) is decremented by 1 after each repetition of the string instruction; therefore, CX must be initialized to the number of repetitions desired before the string instruction is executed. If CX is 0, the string instruction is not executed, and control goes to the following instruction.

Section 2.10 contains examples that illustrate the use of all the string instructions.

REP/REPE/REPZ/REPNE/REPNZ

Repeat, Repeat While Equal, Repeat While Zero, Repeat While Not Equal and Repeat While Not Zero are five mnemonics for two forms of the prefix byte that controls repetition of a subsequent string instruction. The different mnemonics are provided to improve program clarity. The repeat prefixes do not affect the flags.

REP is used in conjunction with the MOVS (Move String) and STOS (Store String) instructions and is interpreted as "repeat while not end-of-string" (CX not 0). REPE and REPZ operate identically and are physically the same prefix byte as REP. These instructions are used with the CMPS (Compare String) and SCAS (Scan String) instructions and require ZF (posted by these instructions) to be set before initiating the next repetition. REPNE and REPNZ are two mnemonics for the same prefix byte. These instructions function the same as REPE and REPZ except that the zero flag must be cleared or the repetition is terminated. Note that ZF does not need to be initialized before executing the repeated string instruction.

Repeated string sequences are interruptable; the processor will recognize the interrupt before processing the next string element. System interrupt processing is not affected in any way. Upon return from the interrupt, the repeated operation is resumed from the point of interruption. Note, however, that execution does *not* resume properly if a second or third prefix (i.e., segment override or LOCK) has been specified in addition to any of the repeat prefixes. The processor "remembers" only one prefix in effect at the time of the interrupt, the prefix that immediately precedes the string instruction. After returning from the interrupt, processing resumes at this point, but any additional prefixes specified are not in effect. If more than one prefix must be used with a string instruction, interrupts may be disabled for the duration of the repeated execution. However, this will not prevent a non-maskable interrupt from being recognized. Also, the time that the system is unable to respond to interrupts may be unacceptable if long strings are being processed.

MOVS *destination-string,source-string*

MOVS (Move String) transfers a byte or a word from the source string (addressed by SI) to the destination string (addressed by DI) and updates SI and DI to point to the next string element. When used in conjunction with REP, MOVS performs a memory-to-memory block transfer.

MOVSB/MOVSW

These are alternate mnemonics for the move string instruction. These mnemonics are coded without operands; they explicitly tell the assembler that a byte string (MOVSB) or a word string (MOVSW) is to be moved (when MOVS is coded, the assembler determines the string type from the attributes of the operands). These mnemonics are useful when the assembler cannot determine the attributes of a string, e.g., a section of code is being moved.

CMPS *destination-string,source-string*

CMPS (Compare String) subtracts the destination byte or word (addressed by DI) from the source byte or word (addressed by SI). CMPS affects the flags but does not alter either operand, updates SI and DI to point to the next string element and updates AF, CF, OF, PF, SF and ZF to reflect the relationship of the destination element to the source element. For example, if a JG (Jump if Greater) instruction follows CMPS, the jump is taken if the destination element is greater than the source element. If CMPS is prefixed with REPE

or REPZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are equal (ZF = 1)." If CMPS is preceded by REPNE or REPNZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are not equal (ZF = 0)." Thus, CMPS can be used to find matching or differing string elements.

SCAS *destination-string*

SCAS (Scan String) subtracts the destination string element (byte or word) addressed by DI from the content of AL (byte string) or AX (word string) and updates the flags, but does not alter the destination string or the accumulator. SCAS also updates DI to point to the next string element and AF, CF, OF, PF, SF and ZF to reflect the relationship of the scan value in AL/AX to the string element. If SCAS is prefixed with REPE or REPZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element = scan-value (ZF = 1)." This form may be used to scan for departure from a given value. If SCAS is prefixed with REPNE or REPNZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element is not equal to scan-value (ZF = 0)." This form may be used to locate a value in a string.

LODS *source-string*

LODS (Load String) transfers the byte or word string element addressed by SI to register AL or AX, and updates SI to point to the next element in the string. This instruction is not ordinarily repeated since the accumulator would be overwritten by each repetition, and only the last element would be retained. However, LODS is very useful in software loops as part of a more complex string function built up from string primitives and other instructions.

STOS *destination-string*

STOS (Store String) transfers a byte or word from register AL or AX to the string element addressed by DI and updates DI to point to the next location in the string. As a repeated operation, STOS provides a convenient way to initialize a string to a constant value (e.g., to blank out a print line).

Program Transfer Instructions

The sequence of execution of instructions in an 8086/8088 program is determined by the content of the code segment register (CS) and the instruction pointer (IP). The CS register contains the base address of the current code segment, the 64k portion of memory from which instructions are presently being fetched. The IP is used as an offset from the beginning of the code segment; the combination of CS and IP points to the memory location from which the next instruction is to be fetched. (Recall that under most operating conditions, the next instruction to be *executed* has already been fetched from memory and is waiting in the CPU instruction queue.) The program transfer instructions operate on the instruction pointer and on the CS register; changing the content of these causes normal sequential execution to be altered. When a program transfer occurs, the queue no longer contains the correct instruction, and the BIU obtains the next instruction from memory using the new IP and CS values, passes the instruction directly to the EU, and then begins refilling the queue from the new location.

Four groups of program transfers are available in the 8086/8088 (see table 2-14): unconditional transfers, conditional transfers, iteration control instructions and interrupt-related instructions. Only the interrupt-related instructions affect any CPU flags. As will be seen, however, the execution of many of the program transfer instructions is affected by the states of the flags.

Unconditional Transfers

The unconditional transfer instructions may transfer control to a target instruction within the current code segment (intrasegment transfer) or to a different code segment (intersegment transfer). (The ASM-86 assembler terms an intrasegment target NEAR and an intersegment target FAR.) The transfer is made unconditionally any time the instruction is executed.

CALL *procedure-name*

CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL. The

Table 2-14. Program Transfer Instructions

UNCONDITIONAL TRANSFERS	
CALL	Call procedure
RET	Return from procedure
JMP	Jump
CONDITIONAL TRANSFERS	
JA/JNBE	Jump if above/not below nor equal
JAE/JNB	Jump if above or equal/not below
JB/JNAE	Jump if below/not above nor equal
JBE/JNA	Jump if below or equal/not above
JC	Jump if carry
JE/JZ	Jump if equal/zero
JG/JNLE	Jump if greater/not less nor equal
JGE/JNL	Jump if greater or equal/not less
JL/JNGE	Jump if less/not greater nor equal
JLE/JNG	Jump if less or equal/not greater
JNC	Jump if not carry
JNE/JNZ	Jump if not equal/not zero
JNO	Jump if not overflow
JNP/JPO	Jump if not parity/parity odd
JNS	Jump if not sign
JO	Jump if overflow
JP/JPE	Jump if parity/parity even
JS	Jump if sign
ITERATION CONTROLS	
LOOP	Loop
LOOPE/LOOPZ	Loop if equal/zero
LOOPNE/LOOPNZ	Loop if not equal/not zero
JCXZ	Jump if register CX = 0
INTERRUPTS	
INT	Interrupt
INTO	Interrupt if overflow
IRET	Interrupt return

assembler generates a different type of CALL instruction depending on whether the programmer has defined the procedure name as NEAR or FAR. For control to return properly, the type of CALL instruction must match the type of RET instruction that exits from the procedure. (The potential for a mismatch exists if the procedure and the CALL are contained in separately assembled programs.) Different forms of the CALL instruction allow the address of the target procedure to be obtained from the instruction itself (direct CALL) or from a memory location or register referenced by the instruction (indirect CALL). In the following descriptions, bear in mind that the processor automatically adjusts IP to point to the next instruction to be *executed* before saving it on the stack.

For an intrasegment direct CALL, SP (the stack pointer) is decremented by two and IP is pushed onto the stack. The relative displacement (up to ±32k) of the target procedure from the CALL instruction is then added to the instruction pointer. This form of the CALL instruction is "self-relative" and is appropriate for position-independent (dynamically relocatable) routines in which the CALL and its target are in the same segment and are moved together.

An intrasegment indirect CALL may be made through memory or through a register. SP is decremented by two and IP is pushed onto the stack. The offset of the target procedure is obtained from the memory word or 16-bit general register referenced in the instruction and replaces IP.

For an intersegment direct CALL, SP is decremented by two, and CS is pushed onto the stack. CS is replaced by the segment word contained in the instruction. SP again is decremented by two. IP is pushed onto the stack and is replaced by the offset word contained in the instruction.

For an intersegment indirect CALL (which only may be made through memory), SP is decremented by two, and CS is pushed onto the stack. CS is then replaced by the content of the second word of the doubleword memory pointer referenced by the instruction. SP again is decremented by two, and IP is pushed onto the stack and is replaced by the content of the first word of the doubleword pointer referenced by the instruction.

RET *optional-pop-value*

RET (Return) transfers control from a procedure back to the instruction following the CALL that activated the procedure. The assembler generates an intrasegment RET if the programmer has defined the procedure NEAR, or an intersegment RET if the procedure has been defined as FAR. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by two. If RET is intersegment, the word at the new top of stack is popped into the CS register, and SP is again incremented by two. If an optional pop value has been specified, RET adds that value to SP. This feature may be used to discard parameters pushed onto the stack before the execution of the CALL instruction.

JMP *target*

JMP unconditionally transfers control to the target location. Unlike a CALL instruction, JMP does not save any information on the stack, and no return to the instruction following the JMP is expected. Like CALL, the address of the target operand may be obtained from the instruction itself (direct JMP) or from memory or a register referenced by the instruction (indirect JMP).

An intrasegment direct JMP changes the instruction pointer by adding the relative displacement of the target from the JMP instruction. If the assembler can determine that the target is within 127 bytes of the JMP, it automatically generates a two-byte form of this instruction called a SHORT JMP; otherwise, it generates a NEAR JMP that can address a target within ±32k. Intrasegment direct JMPS are self-relative and are appropriate in position-independent (dynamically relocatable) routines in which the JMP and its target are in the same segment and are moved together.

An intrasegment indirect JMP may be made either through memory or through a 16-bit general register. In the first case, the content of the word referenced by the instruction replaces the instruction pointer. In the second case, the new IP value is taken from the register named in the instruction.

An intersegment direct JMP replaces IP and CS with values contained in the instruction.

An intersegment indirect JMP may be made only through memory. The first word of the doubleword pointer referenced by the instruction replaces IP, and the second word replaces CS.

Conditional Transfers

The conditional transfer instructions are jumps that may or may not transfer control depending on the state of the CPU flags at the time the instruction is executed. These 18 instructions (see table 2-15) each test a different combination of flags for a condition. If the condition is "true," then control is transferred to the target specified in the instruction. If the condition is "false," then control passes to the instruction that follows the conditional jump. All conditional jumps are SHORT, that is, the target must be in the current code segment and within −128 to +127 bytes of the first byte of the next instruction (JMP 00H jumps to the first byte of the next instruction). Since the jump is made by adding the relative displacement of the target to the instruction pointer, all conditional jumps are self-relative and are appropriate for position-independent routines.

Iteration Control

The iteration control instructions can be used to regulate the repetition of software loops. These instructions use the CX register as a counter. Like the conditional transfers, the iteration control instructions are self-relative and may only transfer to targets that are within −128 to +127 bytes of themselves, i.e., they are SHORT transfers.

LOOP *short-label*

LOOP decrements CX by 1 and transfers control to the target operand if CX is not 0; otherwise the instruction following LOOP is executed.

LOOPE/LOOPZ *short-label*

LOOPE and LOOPZ (Loop While Equal and Loop While Zero) are different mnemonics for the same instruction (similar to the REPE and

Table 2-15. Interpretation of 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
JG/JNLE	((SF XOR OF) OR ZF)=0	greater/not less nor equal
JGE/JNL	(SF XOR OF)=0	greater or equal/not less
JL/JNGE	(SF XOR OF)=1	less/not greater nor equal
JLE/JNG	((SF XOR OF) OR ZF)=1	less or equal/not greater
JNC	CF=0	not carry
JNE/JNZ	ZF=0	not equal/not zero
JNO	OF=0	not overflow
JNP/JPO	PF=0	not parity/parity odd
JNS	SF=0	not sign
JO	OF=1	overflow
JP/JPE	PF=1	parity/parity equal
JS	SF=1	sign

Note: "above" and "below" refer to the relationship of two unsigned values;
"greater" and "less" refer to the relationship of two signed values.

REPZ repeat prefixes). CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and if ZF is set; otherwise the instruction following LOOPE/LOOPZ is executed.

LOOPNE/LOOPNZ *short-label*

LOOPNE and LOOPNZ (Loop While Not Equal and Loop While Not Zero) are also synonyms for the same instruction. CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and if ZF is clear; otherwise the next sequential instruction is executed.

JCXZ *short-label*

JCXZ (Jump If CX Zero) transfers control to the target operand if CX is 0. This instruction is useful at the beginning of a loop to bypass the loop if CX has a zero value, i.e., to execute the loop zero times.

Interrupt Instructions

The interrupt instructions allow interrupt service routines to be activated by programs as well as by external hardware devices. The effect of software interrupts is similar to hardware-initiated interrupts. However, the processor does not execute an interrupt acknowledge bus cycle if the interrupt originates in software or with an NMI. The effect of the interrupt instructions on the flags is covered in the description of each instruction.

INT *interrupt-type*

INT (Interrupt) activates the interrupt procedure specified by the interrupt-type operand. INT decrements the stack pointer by two, pushes the flags onto the stack, and clears the trap (TF) and interrupt-enable (IF) flags to disable single-step and maskable interrupts. The flags are stored in the format used by the PUSHF instruction. SP is decremented again by two, and the CS register is pushed onto the stack. The address of the interrupt pointer is calculated by multiplying interrupt-type by four; the second word of the interrupt pointer replaces CS. SP again is decremented by two, and IP is pushed onto the stack and is replaced by the first word of the interrupt pointer. If interrupt-type = 3, the assembler generates a short (1 byte) form of the instruction, known as the breakpoint interrupt.

Software interrupts can be used as "supervisor calls," i.e., requests for service from an operating system. A different interrupt-type can be used for each type of service that the operating system could supply for an application program. Software interrupts also may be used to check out interrupt service procedures written for hardware-initiated interrupts.

INTO

INTO (Interrupt on Overflow) generates a software interrupt if the overflow flag (OF) is set; otherwise control proceeds to the following instruction without activating an interrupt procedure. INTO addresses the target interrupt procedure (its type is 4) through the interrupt pointer at location 10H; it clears the TF and IF flags and otherwise operates like INT. INTO may be written following an arithmetic or logical operation to activate an interrupt procedure if overflow occurs.

IRET

IRET (Interrupt Return) transfers control back to the point of interruption by popping IP, CS and the flags from the stack. IRET thus affects all flags by restoring them to previously saved values. IRET is used to exit any interrupt procedure, whether activated by hardware or software.

Processor Control Instructions

These instructions (see table 2-16) allow programs to control various CPU functions. One group of instructions updates flags, and another group is used primarily for synchronizing the 8086 or 8088 with external events. A final instruction causes the CPU to do nothing. Except for the flag operations, none of the processor control instructions affect the flags.

Flag Operations

CLC

CLC (Clear Carry flag) zeroes the carry flag (CF) and affects no other flags. It (and CMC and STC) is useful in conjunction with the RCL and RCR instructions.

Table 2-16. Processor Control Instructions

FLAG OPERATIONS	
STC	Set carry flag
CLC	Clear carry flag
CMC	Complement carry flag
STD	Set direction flag
CLD	Clear direction flag
STI	Set interrupt enable flag
CLI	Clear interrupt enable flag
EXTERNAL SYNCHRONIZATION	
HLT	Halt until interrupt or reset
WAIT	Wait for $\overline{\text{TEST}}$ pin active
ESC	Escape to external processor
LOCK	Lock bus during next instruction
NO OPERATION	
NOP	No operation

CMC

CMC (Complement Carry flag) "toggles" CF to its opposite state and affects no other flags.

STC

STC (Set Carry flag) sets CF to 1 and affects no other flags.

CLD

CLD (Clear Direction flag) zeroes DF causing the string instructions to auto-increment the SI and/or DI index registers. CLD does not affect any other flags.

STD

STD (Set Direction flag) sets DF to 1 causing the string instructions to auto-decrement the SI and/or DI index registers. STD does not affect any other flags.

CLI

CLI (Clear Interrupt-enable flag) zeroes IF. When the interrupt-enable flag is cleared, the 8086 and 8088 do not recognize an external interrupt request that appears on the INTR line; in other words maskable interrupts are disabled. A non-maskable interrupt appearing on the NMI line, however, is honored, as is a software interrupt. CLI does not affect any other flags.

STI

STI (Set Interrupt-enable flag) sets IF to 1, enabling processor recognition of maskable interrupt requests appearing on the INTR line. Note however, that a pending interrupt will not actually be recognized until the instruction following STI has executed. STI does not affect any other flags.

External Synchronization

HLT

HLT (Halt) causes the 8086/8088 to enter the halt state. The processor leaves the halt state upon activation of the RESET line, upon receipt of a non-maskable interrupt request on NMI, or, if interrupts are enabled, upon receipt of a maskable interrupt request on INTR. HLT does not affect any flags. It may be used as an alternative to an endless software loop in situations where a program must wait for an interrupt.

WAIT

WAIT causes the CPU to enter the wait state while its $\overline{\text{TEST}}$ line is not active. WAIT does not affect any flags. This instruction is described more completely in section 2.5.

ESC *external-opcode, source*

ESC (Escape) provides a means for an external processor to obtain an opcode and possibly a memory operand from the 8086 or 8088. The external opcode is a 6-bit immediate constant that the assembler encodes in the machine instruction it builds (see table 2-26). An external processor may monitor the system bus and capture this opcode when the ESC is fetched. If the source operand is a register, the processor does nothing. If the source operand is a memory variable, the processor obtains the operand from memory and discards it. An external processor may capture the memory operand when the processor reads it from memory.

LOCK

LOCK is a one-byte prefix that causes the 8086/8088 (configured in maximum mode) to assert its bus $\overline{\text{LOCK}}$ signal while the following instruction executes. LOCK does not affect any flags. See section 2.5 for more information on LOCK.

No Operation

NOP

NOP (No Operation) causes the CPU to do nothing. NOP does not affect any flags.

Instruction Set Reference Information

Table 2-21 provides detailed operational information for the 8086/8088 instruction set. The information is presented from the point of view of utility to the assembly language programmer. Tables 2-17, 2-18 and 2-19 explain the symbols used in table 2-21. Machine language instruction encoding and decoding information is given in Chapter 4.

Instruction timings are presented as the number of clock periods required to execute a particular form (register-to-register, immediate-to-memory, etc.) of the instruction. If a system is running with a 5 MHz maximum clock, the maximum clock period is 200 ns; at 8 MHz, the clock period is 125 ns. Where memory operands are used, "+EA" denotes a variable number of additional clock periods needed to calculate the operand's effective address (discussed in section 2.8). Table 2-20 lists all effective address calculation times.