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.

When the EU is ready to execute an instruction, it fetches the instruction object code byte from the BIU's instruction queue and then executes the instruction. If the queue is empty when the EU is ready to fetch an instruction byte, the EU waits for the instruction byte to be fetched. If a memory location or I/O port must be accessed during the execution of an instruction, the EU requests the BIU to perform the required bus cycle.

BUS INTERFACE UNIT

The 8086 and 8088 BIU's are functionally identical, but are implemented differently to match the structure and performance characteristics of their respective buses. Data is transferred between the CPU and memory or I/O devices upon demand from the EU. The BIU executes all external bus cycles. This unit consists of the segment and communications registers, the instruction pointer and the instruction object code queue. The BIU combines segment and offset values in a dedicated adder to derive 20-bit addresses, transfers data to and from EU on the ALU data bus and loads or "prefetches" instructions into the queue. These "prefetched" instructions can then be fetched by the EU with a minimum of wait.

During periods when the EU is busy executing instructions, the BIU "looks ahead" and fetches more instructions from memory. These 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 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 automatically reads one byte from the odd address and then resumes fetching two-byte words from the subsequent even addresses.

In 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 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 CPU's have the same complement of eight 16-bit general registers (see Figure 6-5). The general registers are subdivided into two sets of four registers each. These are 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 are always accessed as 16-bit only. The data registers can be used without constraint in most arithmetic and logic operations. In addition, some instructions use certain registers implicitly (see Table 6-1), therefore allowing compact yet powerful encoding.

The pointer and index registers can also be used in most arithmetic and logic operations. All eight general registers fit the definition of an "accumulator" as defined in first and second generation microprocessors. The P & I registers (except for BP) are also used implicitly in some instructions (see Table 6-1).

SEGMENT REGISTERS

The 8086 and 8088 memory space (up to one megabyte) is divided into logical segments of up to 64K bytes each. The CPU has direct access to four segments at a time. The base addresses (starting locations) of these memory segments are contained in the segment registers (see Figure 6-6). The CS register points to the current code segment. Instructions are fetched from the CS segment. The SS register points to the current stack

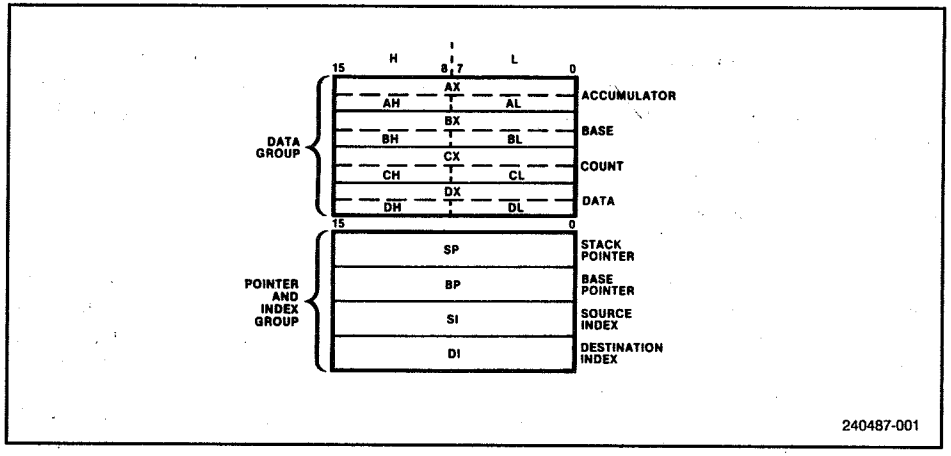

Figure 6-5. General Registers

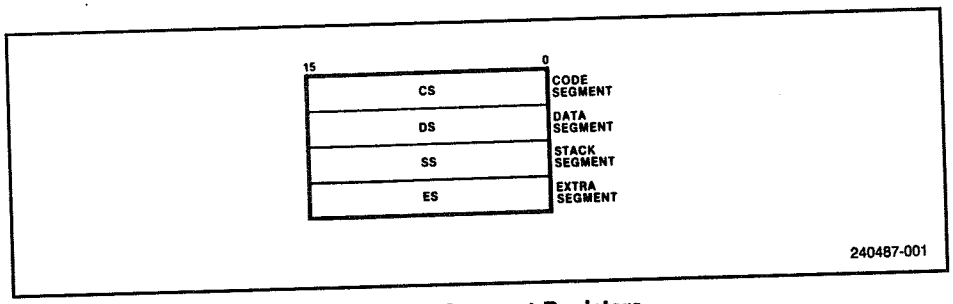

Figure 6-6. Segment Registers

Table 6-1. 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

segment. Stack operations are performed on locations in the SS segment. The DS register points to the current data segment. The DS register 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.

INSTRUCTION POINTER

The 16-bit instruction pointer (IP) is similar 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 is first 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 (see Figure 6-7) that the EU posts to reflect certain properties of the result of an arithmetic or logic operation. A group of instructions is available that allows a program to alter its execution depending on the state of these flags, i.e., 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 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 the 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 (see Figure 6-7) can be set and cleared by programs to alter processor operations:

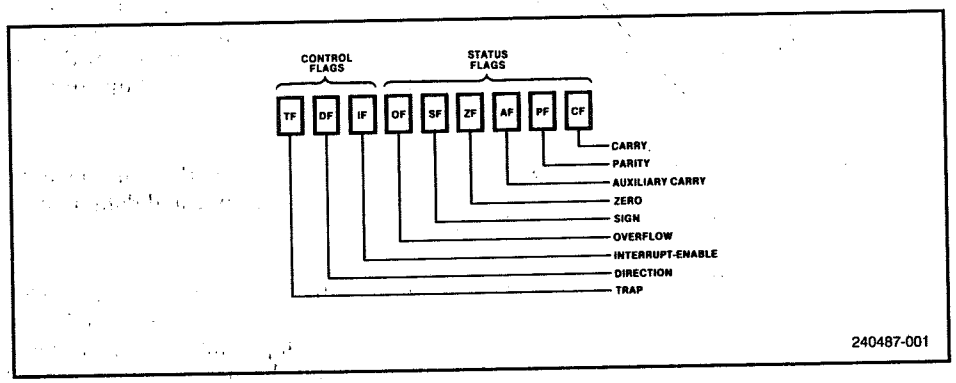

Figure 6-7. Status Flags

1. Setting DF (the direction flag) causes string instructions to auto-decrement; that is, to process strings from the high address to the low address, or from "right to left." Clearing DF causes string instructions to auto-increment, or 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 interrrupt after each instruction, allowing a program to be inspected as it executes instruction by instruction.

MODE SELECTION

Each of the processors has a strap pin (MN/MX*) that defines the function of eight CPU pins in the 8086 and nine pins in the 8088. Connecting MN/MX* to +5V places the CPU in minimum mode. This configuration is designed for small systems (roughly one or two boards) and the CPU provides bus control signals needed by memory and peripherals. When MN/MX* is strapped to ground, the CPU is configured in maximum mode. In this configuration the CPU encodes control signals on three lines. An 82C88 Bus Controller is added to decode the signals for 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.

SEGMENTATION

Programs for the 8086 and 8088 "view" the memory space (one megabyte) as a group of segments that are defined by 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 6-8). 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 6-9). 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.

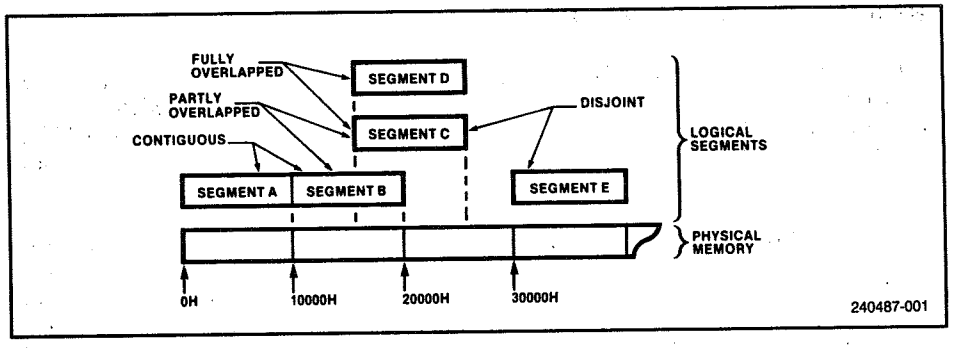

Figure 6-8. Segment Locations in Physical Memory

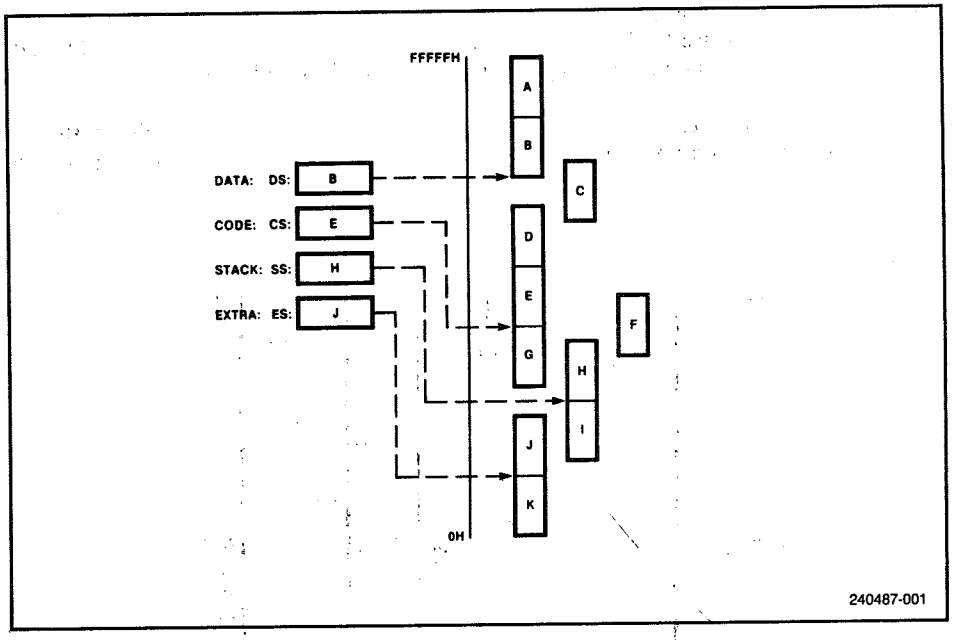

Figure 6-9. Currently Addressable Segments

The segment 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 test 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

In theory, 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 range from 0H to 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. In the example (see Figure 6-10), 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

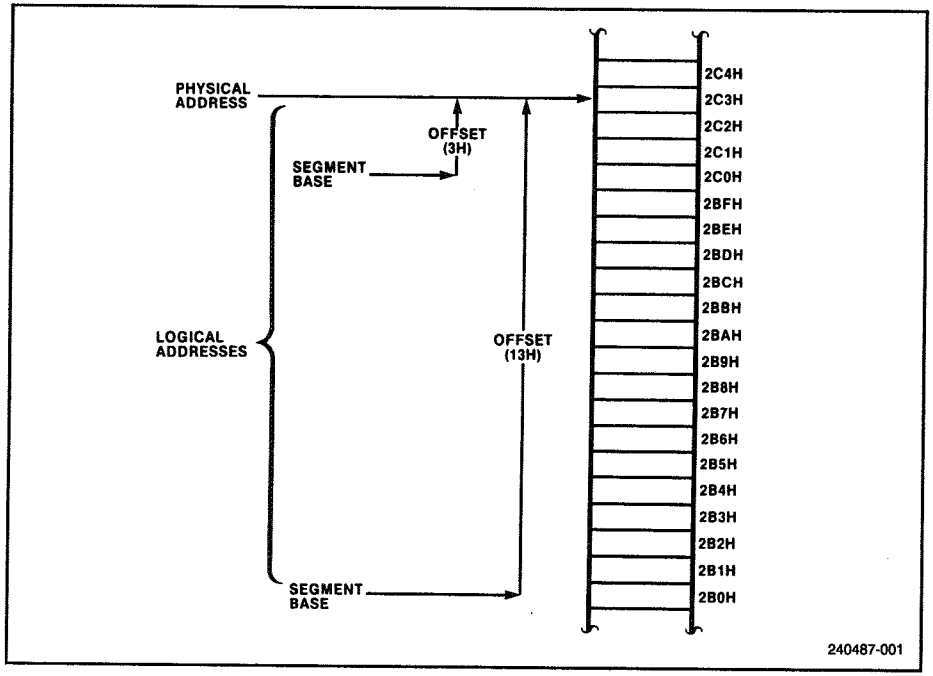

Figure 6-10. Logical and Physical Addresses

the segment base value four bit positions and adding the offset as illustrated in Figure 6-11. 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 6-2). Instructions are always 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).

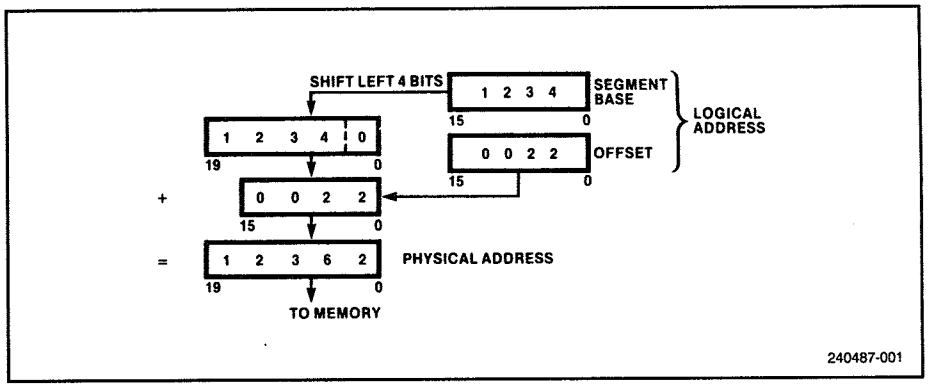

Figure 6-11. Physical Address Generation

Table 6-2. Logical Addresses 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

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 extra segment; its offset is taken from DI, the destination index register. The string instruction 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. Therefore, register BP provides a convenient way to address data on the stack. However, BP can also be used to access data in any of the other currently addressable segments.

The BIU's segment assumptions are a convenience to programmers in most cases. However, it is possible 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 an 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 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 only available in non-adjacent fragments, other program segments can be compacted to free up a continuous space. This process is illustrated graphically in Figure 6-12.

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 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.

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 (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 segment.) 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 stack (TOS). In other words, SP contains the offset of the top of the stack from the stack segment's base address. However, the stack's base address (contained in SS) is not the "bottom" of the stack.

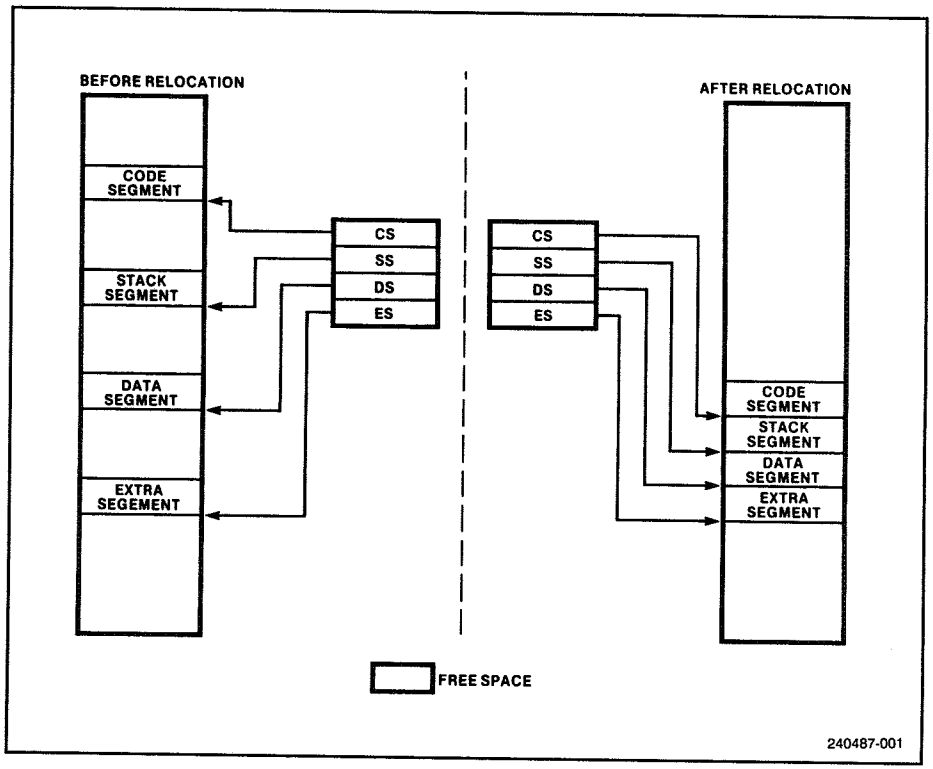

Figure 6-12. Dynamic Code Relocation

Stacks in the 8086 and 8088 are 16 bits wide; instructions that operate on a stack and remove stack items one word a time. An item is pushed onto the stack (see Figure 6-13) by *decrementing* SP by 2 and writing the item at a new TOS. An item is popped off the stack by copying it from TOS and the *incrementing* SP by 2. In other words, the stack goes *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.

RESERVED MEMORY

Two areas in extreme low and high memory (see Figure 6-14) are dedicated to specific processor functions or are reserved by Intel Corporation for use by Intel hardware and software products. The locations are 0H through 7FH (128 bytes) and FFFF0H through FFFFFH (16 bytes). These areas are used for interrupt and system reset processing. 8086 and 8088 application systems do not use these areas for any other purpose. Doing so may make these systems incompatible with future Intel products.

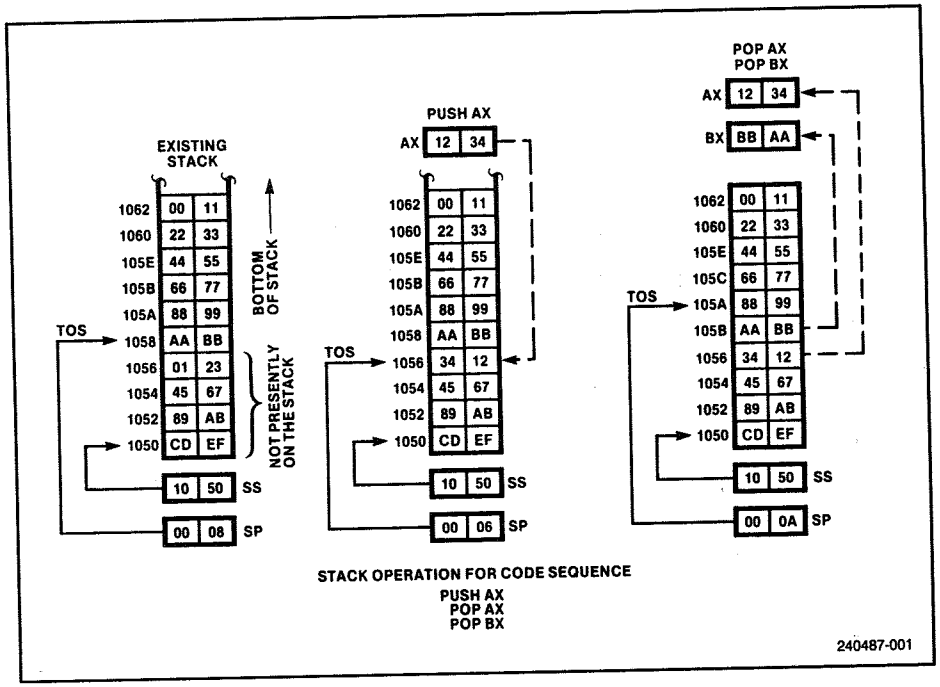

Figure 6-13. Stack Operation

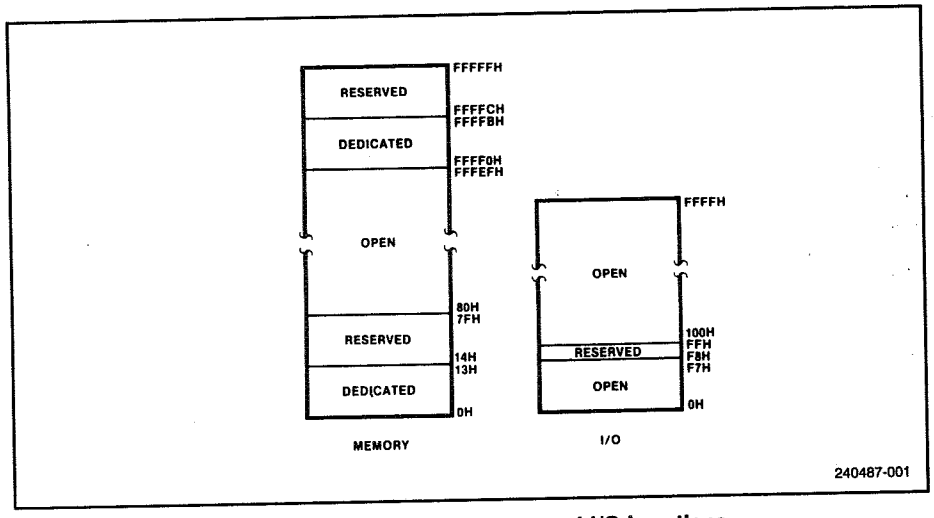

Figure 6-14. Reserved Memory and I/O Locations

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 are also fetched one byte at a time. Although alignment of word operands does not affect the performance of 8088, locating 16-bit data on even addresses will insure maximum throughput if the system is ever transferred to an 8086.

Software Overview

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,
- a group of instructions that can help coordinate the activities of multiprocessing systems.

The following paragraphs provide a description of the instructions by category and a detailed discussion of the various operand addressing modes. In addition, a complete instruction set summary is provided in tabular form which recaps each device instruction by category, and provides timing cycles for each instruction. Information is also described on how to encode and decode machine instructions for any given assembly code instruction.

8086/8088 INSTRUCTION SET

The 8086/8088 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. The exception to this is that immediate values serve as "source" and not "destination" operands. In particular, memory variables may be added to, subtracted from, shifted, compared, and so on, in place, without moving them in and out of register. This saves instructions, registers, and execution time in assembly language programs. In high-level languages, where most variables are memory based, compilers can produce faster and shorter object programs.

The 8086/8088 instruction set can be viewed as existing on two levels. One is the assembly level and the other is the machine level. To the assembly language programmer, the 8086/8088 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 8086/8088 CPU's, however, recognize 28 different MOV machine instructions ("move byte register to memory," "move word immediate to register," etc.).

The two levels of instruction set address two different requirements: efficiency and simplicity. The approximately 300 forms of machine-level instructions make very efficient use of storage. For example, the machine instructions 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. The 8086/88 have eight different machine-level instructions that increment a different 16-bit register. Each of these instructions are only one byte long.

The assembly level instructions simplify the programmers view of the instruction set. The programmer writes one form of an INC (increment) instruction and the ASM-86 assembler examines the operand to determine which machine level instruction to generate. The following paragraphs provide a functional description of the assembly-level instructions.

Data Transfer Instructions

The 8086/8088 instruction set contains 14 data transfer instructions. These instructions move single bytes and words between memory and registers, and also move single bytes and words between the AL or AX registers and I/O ports. Table 6-3 lists the four types of data transfer instructions and their functions.

Data transfer instructions are categorized into four types: 1) general purpose; 2) input/output; 3) address object; and 4) flag transfer. The stack manipulation instructions, which are used for transferring flag contents, and the instructions for loading segment registers are also included in this group. Figure 6-15 shows the flag storage formats. These formats are used primarily by the LAHF instruction when converting 8080/8085 assembly language programs to run on the 8086 or 8088. The address object instructions manipulate the addresses of variables instead of the contents of values of the variables. This is useful for list processing, based variable and string operations.

Table 6-3. 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

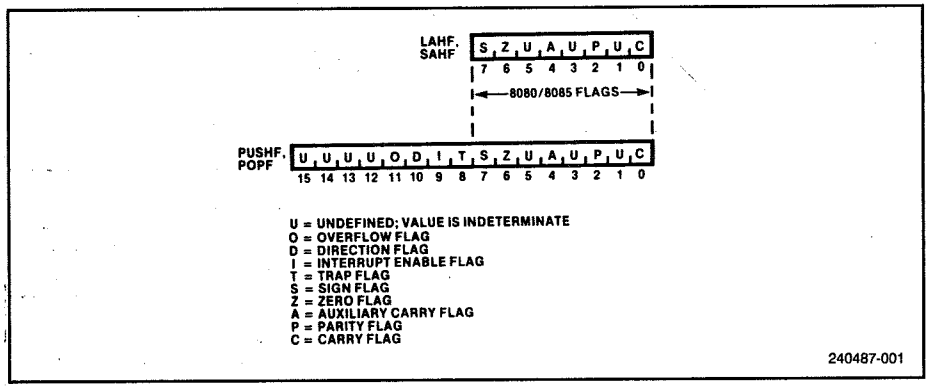

Figure 6-15. Flag Storage Formats

Arithmetic Instructions

The arithmetic instructions (see Table 6-4) perform operations on four types of numbers: 1) unsigned binary; 2) signed binary (integers); 3) unsigned packed decimal; and 4) unsigned unpacked decimal. See Table 6-5. 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 represents valid numbers for the type of instruction being performed. Invalid data may produce unpredictable results.

Arithmetic instructions post certain characteristics of the result of an operation to six flags. Refer to Chapter 3 for a detailed description of the arithmetic instructions and flags.

Table 6-4. 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 6-5. Arithmetic Interpretation of 8-Bit Numbers

Hex	Bit Pattern	Unsigned Binary	Signed Binary	Unpacked Decimal	Packed Decimal
07	00000111	7	+7	7	7
89	10001001	137	−119	invalid	89
C5	11000101	197	−59	invalid	invalid

Bit Manipulation Instructions

The 8086 and 8088 CPU's provide three groups of instructions for manipulating bits within both bytes and word. These three groups are logicals, shifts and rotates. Table 6-6 lists these three groups of bit manipulation instructions with their functions.

a. Logical

The logical instructions include the boolean operators "not," "and," "inclusive or," and "exclusive or." A TEST instruction that sets the flags as a result of a boolean "and" operation, but does not alter either of its operands, is also included.

b. 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 a constant 1, or 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. Logical shifts can be used to isolate bits in bytes or words.

c. Rotates

Bits in bytes and words can also 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 reigster. 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.

Table 6-6. 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

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 many be manipulated with these instructions. Instructions are available to move, compare and scan for a value, as well as moving string elements to and from the accumulator. Table 6-7 lists the string instructions. 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 similarly in many respects (refer to Table 6-8). 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 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. However, the assembler does not use the operand names to address strings. Instead, the contents of register SI (source index) is used as an offset to address the current element of the source string. Also, the contents 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 instructions. The LDS, LES and LEA instructions are useful in performing this function.

Table 6-7. 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 6-8. 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

String instructions automatically update SI and/or DI in anticipation of processing the next string element. Setting DF (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 by 1. 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. 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.

Program Transfer Instructions

The sequence in which instructions are executed in the 8086/8088 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 currently being fetched. The IP points to the memory location from which the next instruction is to be fetched. In most operating conditions, the next instruction to be executed will have already been fetched 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 operation to be altered. When a program transfer occurs, the queue no longer contains the correct instruction. When the BIU obtains the next instruction from memory using the new IP and CS values, it passes the instruction directly to the EU and then begins refilling the queue from the new location.

Four groups of program transfers are available with the 8086/8088 CPU's. See Table 6-9. These are unconditional transfers, conditional transfers, iteration control instructions, and interrupt-related instructions.

a. 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 transfer SHORT or NEAR and an intersegment transfer FAR. The transfer is made unconditionally any time the instruction is executed.

b. 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 6-10) 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 jumps are 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.

c. 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.

d. 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.

Table 6-9. 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

Processor Control Instructions

The processor control instructions (see Table 6-11) 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 to 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.

Table 6-10. 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.

Table 6-11. 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 TEST pin active
ESC	Escape to external processor
LOCK	Lock bus during next instruction
No Operation	
NOP	No operation

OPERAND ADDRESSING MODES

The 8086 and 8088 access instruction operands in many different ways. Operands may be contained in registers, within the instruction itself, in memory, or at I/O ports. Also, the addresses of memory and I/O port operands can be calculated in several different ways. These addressing modes greatly extend the flexibility and convenience of the instruction set. The following paragraphs briefly describe the register and immediate modes of operand addressing, and then provide a detailed description of the memory and I/O addressing modes.

Register and Immediate Operands

Instructions that specify only register operands are generally the most compact and fastest executing of the operand addressing forms. This is because the register operand addresses are encoded in instructions in just a few bits, and because these operands are performed entirely within the CPU (no bus cycles are run). Registers may serve as source operands, destination operands, or both.

Immediate operands are constant data contained in an instruction. The data may be either 8 or 16 bits in length. Immediate operands can be accessed quickly because they are available directly from the instruction queue. Like the register operand, no bus cycles need to be run to obtain an immediate operand. The limitations on immediate operands are that they may only serve as source operands and that they are constant value.

Memory Addressing Modes

Although the EU has direct access to register and immediate operands, memory operands must be transferred to and from the CPU over the bus. When the EU needs to read or write a memory operand, it must pass an offset value to the BIU. The BIU adds the offset to the (shifted) content of a segment register producing a 20-bit physical address and then executes the bus cycle or cycles needed to access the operand.

a. The Effective Address

The offset that the EU calculates for a memory operand is called the operand's effective address of EA. This address is an unsigned 16-bit number that expresses the operand's distance in bytes from the beginning of the segment in which it resides. The EU can calculate the effective address in several different ways. Information encoded in the second byte of the instruction tells the EU how to calculate the effective address of each memory operand. A compiler or assembler derives this information from the statement or instruction written by the programmer. Assembly language programmers have access to all addressing modes.

The EU calculates the EA by summing a displacement, the content of a base register and the content of an index register (see Figure 6-16). Any combination of these three components may be present in a given instruction. This allows a variety of memory addressing modes.

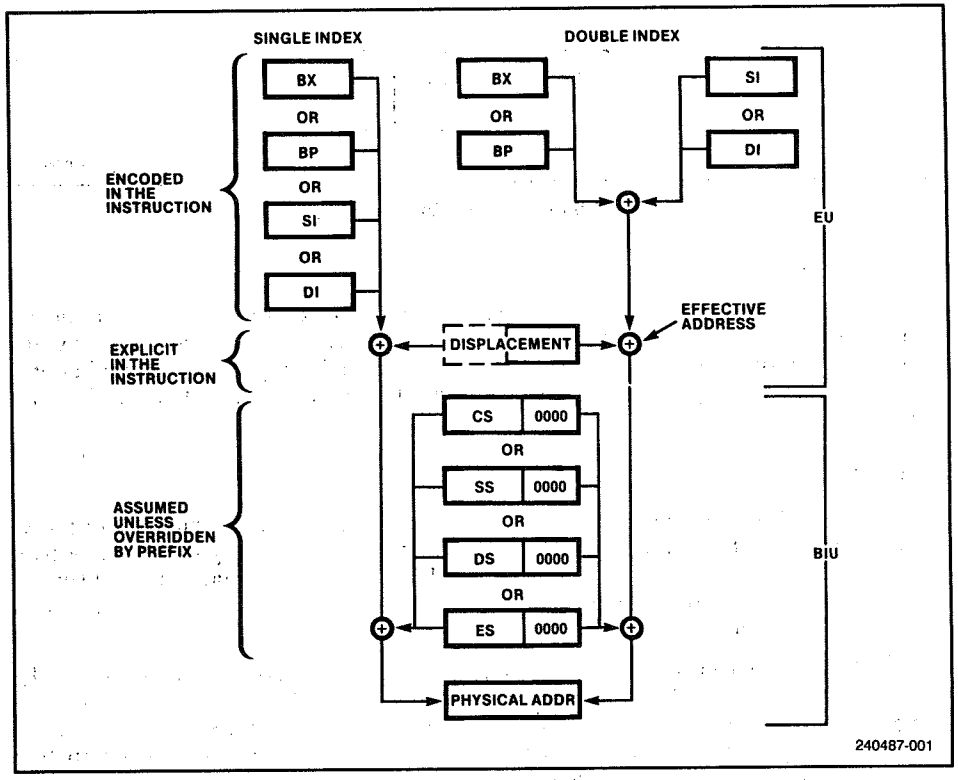

Figure 6-16. Memory Address Computation

The displacement element is an 8- or 16-bit number that is contained in the instruction. The displacement generally is derived from the position of the operand name (a variable or label) in the program. The programmer can also modify this value or explicitly specify the displacement.

A programmer may specify that either BX or BP is to serve as a base register whose content is to be used in the EA computation.

b. Similarly, either SI or DI may be specified as the index register. The displacement value is a constant. The contents of the base and index registers may change during execution. This allows one instruction to access different memory locations as determined by the current values in the base and/or index registers. Effective address calculations with the BP are made using the SS register, by default, although either the DS or the ES registers may be specified instead.

c. Direct Addressing

Direct addressing is the simplest memory addressing mode (see Figure 6-17). No registers are involved and the EA is taken directly from the displacement of the instruction. Direct addressing is typically to access simple variables (scalars).

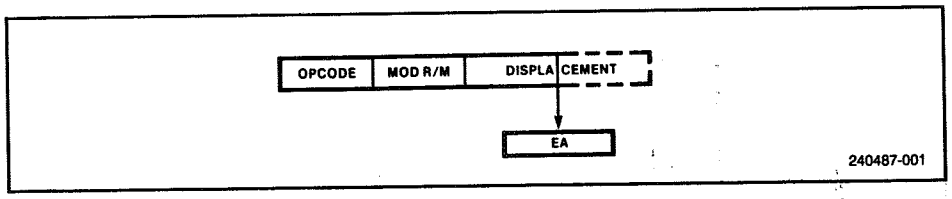

Figure 6-17. Direct Addressing

d. Register Indirect Addressing

The effective address of a memory operand may be taken directly from one of the base or index registers (see Figure 6-18). One instruction can operate on many different memory locations if the value in the base or index register is updated appropriately. Any 16-bit general register may be used for register indirect addressing with the JMP or CALL instructions.

e. Based Addressing

In based addressing (see Figure 6-19), the effective address is the sum of a displacement value and the content of register BX or BP. Specifying register BP as a base register directs the BIU to obtain the operand from the current stack segment (unless a segment override prefix is present). This makes based addressing with BP a very convenient way to access stack data.

Based addressing also provides a simple way to address structures which may be located at different places in memory (see Figure 6-20). A base register can be pointed at the base of the structure and elements of the structure can be addressed by their displacement from the structure base. Different copies of the same structure can be accessed by simply changing the base register.

f. Indexed Address

The effective address is calculated from the sum of a displacement plus the content of an index register (SI or DI) in index addressing (see Figure 6-21). Indexed address is often used to access elements in an array (see Figure 6-22). The displacement locates the beginning of the array, and the value of the index register selects one element (the first element is selected if the index register contains 0). Since all array elements are the same length, simple arithmetic on the index register may select any element.

g. Based Index Addressing

Based index addressing generates an effective address that is the sum of a base register, an index register and a displacement (see Figure 6-23). This mode of addressing is very flexible because two address components can be varied at execution time.

Based index addressing provides a convenient way for a procedure to address an array allocated on a stack (see Figure 6-24). Reigster BP can contain the offset of a reference point on the stack, typically the top of the stack after the procedure has saved registers and allocated local storage. The offset of the beginning of the array from the reference point can be expressed by a displacement value, and the index

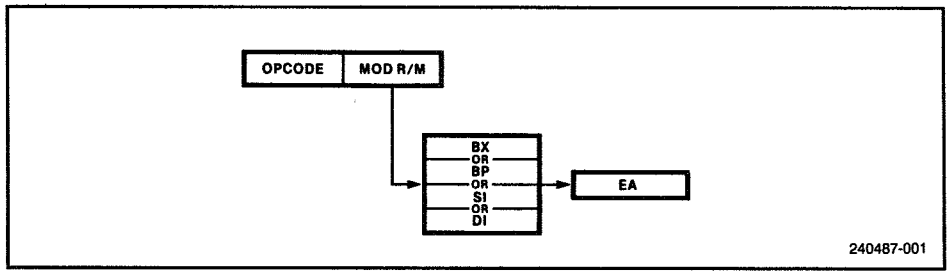

Figure 6-18. Register Indirect Addressing

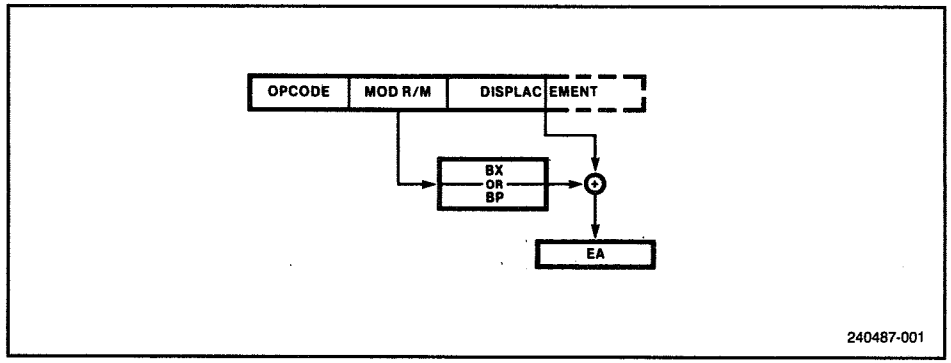

Figure 6-19. Based Addressing

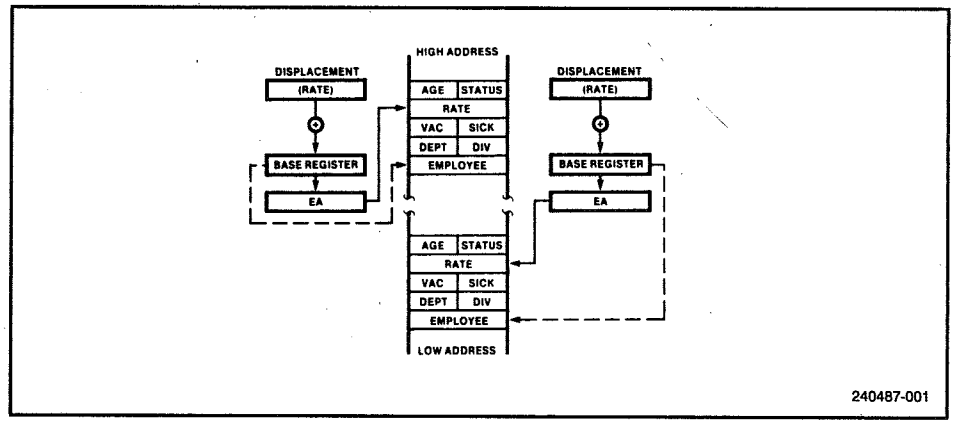

Figure 6-20. Accessing a Structure with Based Addressing

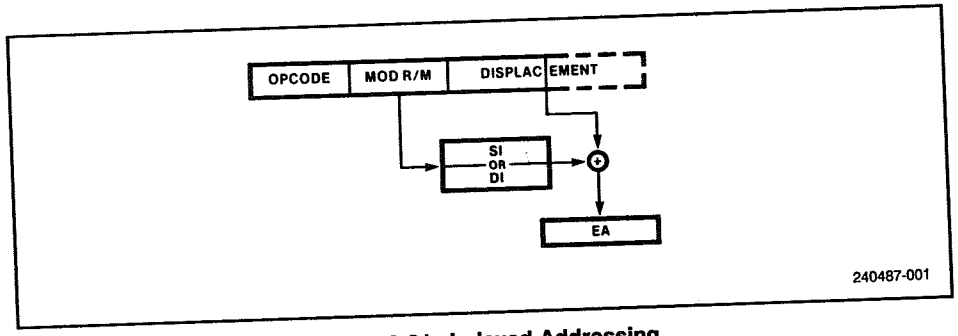

Figure 6-21. Indexed Addressing

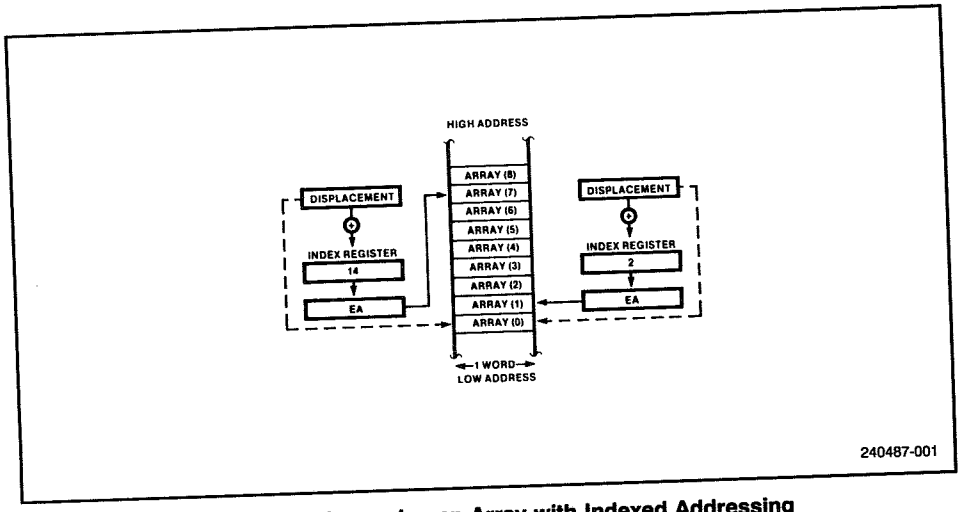

Figure 6-22. Accessing an Array with Indexed Addressing

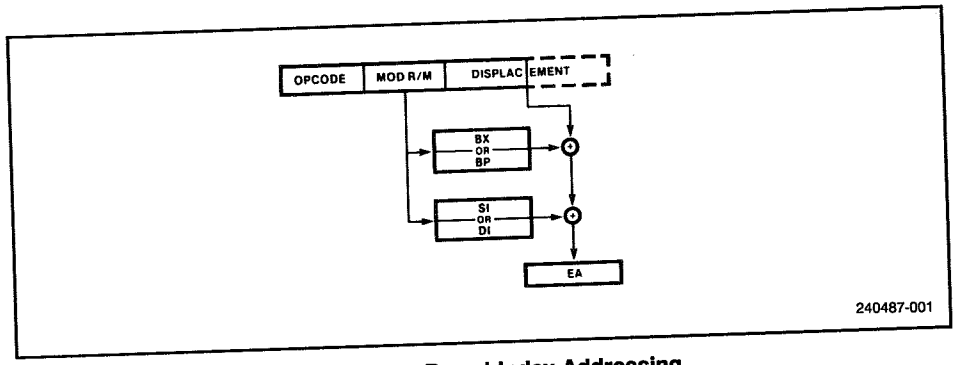

Figure 6-23. Based Index Addressing

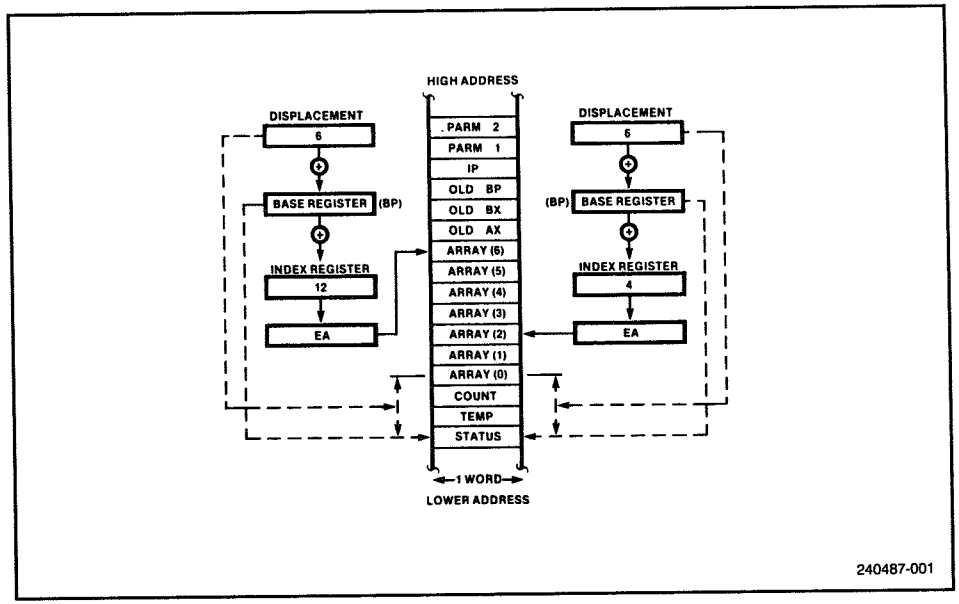

Figure 6-24. Accessing a Stacked Array with Based Index Addressing

register can be used to access individual array elements. Arrays contained in structures and matrices (two-dimensional arrays) can also be accessed with based indexed addressing.

h. String Addressing

String instructions do not use the normal memory addressing modes to access operands. Instead, the index registers are used implicitly (see Figure 6-25). When a string instruction is executed, SI is assumed to point to the first byte or word of the source string. DI is assumed to point at the first byte or word of the destination string. In a repeated string operation, the CPU's automatically adjust SI and DI to obtain subsequent bytes or words. Note that for string instructions DS is the default segment register to SI and ES is the default segment register for DI. This allows string instructions to easily operate on data located anywhere within the one megabyte address space.

I/O Port Addressing

Any of the memory operand addressing modes may be used to access an I/O port if the port is memory mapped. For example, a group of terminals can be accessed as an "array." String instructions can also be used to transfer data to memory-mapped ports with an appropriate hardware interface.

Two different addressing modes can be used to access ports located in the I/O space (see Figure 6-26). The port number is an 8-bit immediate operand for direct addressing. This

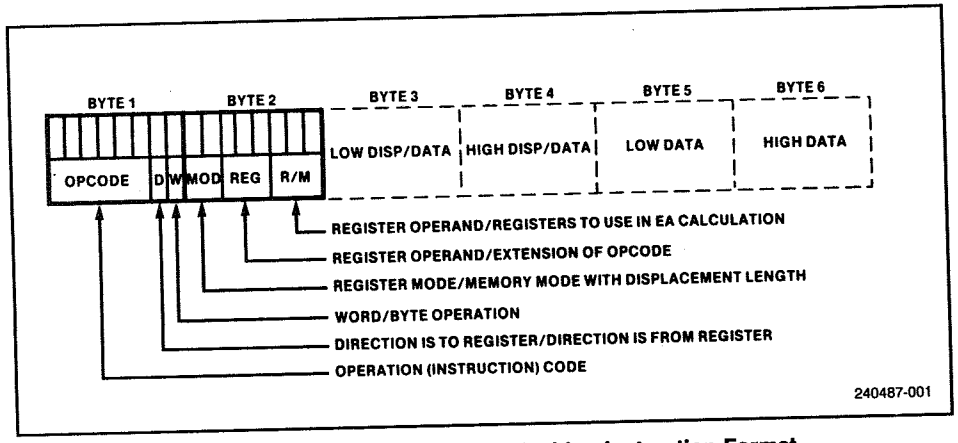

Figure 6-27. Typical 8086/8088 Machine Instruction Format

One of three additional single-bit fields, S, V or Z, appears in some instruction formats (refer to Table 6-17). S, in conjunction with W, indicates the sign extension of immediate fields in arithmetic instructions. V distinguishes between single- and variable-bit shifts and rotates. Z is a compare bit with the zero flag in conditional repeat and loop instructions.

The second byte of the instruction usually identifies the instruction's operands. The MOD (mode) field indicates whether one of the operands is in memory or whether both operands are registers (refer to Table 6-18). The REG (register) field identifies a register that is one of the instruction operands (refer to Table 6-19). In a number of instructions, particularly the immediate-to-memory variety, REG is used as an extension of the opcode to identify the type of operation. The encoding of the R/M (register/memory) field (refer to Table 6-20) depends on how the mode field is set. If MOD = 11 (register-to-register mode), then R/M identifies the second register operand. If MOD selects memory mode, then R/M indicates how the effective address of the memory operand is to be calculated.

Bytes 3 through 6 of an instruction are optional fields that usually contain the displacement value of a memory operand and/or the actual value of an immediate constant operand.

The displacement value may contain one or two bytes; the language translators generate one byte whenever possible. The MOD field indicates how many displacement bytes are present. Following Intel convention, if the displacement is two bytes, the most-significant byte is stored second in the instruction. If the displacement is only a single byte, the 8086 or 8088 automatically sign-extends this quantity to 16-bits before using the information in further address calculations. Immediate values always follow any displacement values that may be present. The second byte of a two-byte immediate value is the most significant.

Table 6-17. Single-Bit Field Encoding

Field	Value	Function
S	0	No sign extension
	1	Sign extend 8-bit immediate data to 16 bits if W=1
W	0	Instruction operates on byte data
	1	Instruction operates on word data
D	0	Instruction source is specified in REG field
	1	Instruction destination is specified in REG field
V	0	Shift/rotate count is one
	1	Shift/rotate count is specified in CL register
Z	0	Repeat/loop while zero flag is clear
	1	Repeat/loop while zero flag is set

Table 6-18. Mode (MOD) Field Encoding

Code	Explanation
00	Memory Mode, no displacement follows*
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

*Except when R/M = 110, then 16-bit displacement follows

Table 6-19. REG (Register) Field Encoding

REG	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Table 6-20. Register/Memory Field Encoding

MOD = 11			Effective Address Calculation			
R/M	W = 0	W = 1	R/M	MOD = 00	MOD = 01	MOD = 10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI) + D8	(DI) + D16
110	DH	SI	110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX)	(BX) + D8	(BX) + D16

Table 6-22 lists the instruction encodings for all 8086/8088 instructions. This table can be used to predict the machine encoding of any ASM-86 instruction. Table 6-23 lists the 8086/8088 machine instructions in order by the binary value of their first byte. This table can be used to decode any machine instruction from its binary representation. Table 6-21 is a key to the abbreviations used in Table 6-22 and 6-23. Figure 6-28 is a more compact instruction decoding guide.

Table 6-21. Key to Machine Instruction Encoding and Decoding

Identifier	Explanation
MOD	Mode field; described in this chapter.
REG	Register field; described in this chapter.
R/M	Register/Memory field; described in this chapter.
SR	Segment register code: 00 = ES, 01 = CS, 10 = SS, 11 = DS.
W, S, D, V, Z	Single-bit instruction fields; described in this chapter.
DATA-8	8-bit immediate constant.
DATA-SX	8-bit immediate value that is automatically sign-extended to 16-bits before use.
DATA-LO	Low-order byte of 16-bit immediate constant.
DATA-HI	High-order byte of 16-bit immediate constant.
(DISP-LO)	Low-order byte of optional 8- or 16-bit unsigned displacement; MOD indicates if present.
(DISP-HI)	High-order byte of optional 16-bit unsigned displacement; MOD indicates if present.
IP-LO	Low-order byte of new IP value.
IP-HI	High-order byte of new IP value.
CS-LO	Low-order byte of new CS value.
CS-HI	High-order byte of new CS value.
IP-INC8	8-bit signed increment to instruction pointer.
IP-INC-LO	Low-order byte of signed 16-bit instruction pointer increment.
IP-INC-HI	High-order byte of signed 16-bit instruction pointer increment.
ADDR-LO	Low-order byte of direct address (offset) of memory operand; EA not calculated.
ADDR-HI	High-order byte of direct address (offset) of memory operand; EA not calculated.
— —	Bits may contain any value.
XXX	First 3 bits of ESC opcode.
YYY	Second 3 bits of ESC opcode.
REG8	8-bit general register operand.
REG16	16-bit general register operand.
MEM8	8-bit memory operand (any addressing mode).
MEM16	16-bit memory operand (any addressing mode).
IMMED8	8-bit immediate operand.
IMMED16	16-bit immediate operand.
SEGREG	Segment register operand.
DEST-STR8	Byte string addressed by DI.
SRC-STR8	Byte string addressed by SI.
DEST-STR16	Word string addressed by DI.
SRC-STR16	Word string addressed by SI.
SHORT-LABEL	Label within ±127 bytes of instruction.
NEAR-PROC	Procedure in current code segment.
FAR-PROC	Procedure in another code segment.
NEAR-LABEL	Label in current code segment but farther than −128 to +127 bytes from instruction.
FAR-LABEL	Label in another code segment.
SOURCE-TABLE	XLAT translation table addressed by BX.
OPCODE	ESC opcode operand.
SOURCE	ESC register or memory operand.

Table 6-22. 8086/8088 Instruction Encoding