CHAPTER 1

THE x86 MICROPROCESSOR

OBJECTIVES

Upon completion of this chapter, you will be able to:

- >> **Describe the Intel family of microprocessors from the 8085 to the Pentium in terms of bus size, physical memory, and special features**
- >> **Explain the function of the EU (execution unit) and BIU (bus interface unit)**
- >> **Describe pipelining and how it enables the CPU to work faster**
- >> **List the registers of the 8086**
- >> **Code simple MOV and ADD instructions and describe the effect of these instructions on their operands**
- >> **State the purpose of the code segment, data segment, stack segment, and extra segment**
- >> **Explain the difference between a logical address and a physical address**
- >> **Describe the "little endian" storage convention of x86 microprocessors**
- >> **State the purpose of the stack**
- >> **Explain the function of PUSH and POP instructions**
- >> **List the bits of the flag register and briefly state the purpose of each bit**
- >> **Demonstrate the effect of ADD instructions on the flag register**
- >> **List the addressing modes of the 8086 and recognize examples of each mode**
- >> **Know how to use flowcharts and pseudocode in program development**

This chapter examines the architecture of the 8086 with some examples of Assembly language programming. Section 1.1 gives a history of the evolution of Intel's family of x86 microprocessors. An overview of the internal workings of 8086 microprocessors is given in Section 1.2. An introduction to 8086 Assembly language programming is covered in Section 1.3. Sections 1.4 and 1.5 cover code and stack segments respectively and show how physical addresses are generated. Section 1.6 explores the flag register and its use in Assembly language programming. Finally, Section 1.7 describes in detail the addressing modes of the 8086.

SECTION 1.1: BRIEF HISTORY OF THE x86 FAMILY

In this section we trace the evolution of Intel's family of microprocessors from the late 1970s, when the personal computer had not yet found widespread acceptance, to the powerful microcomputers widely in use today.

Evolution from 8080/8085 to 8086

In 1978, Intel Corporation introduced a 16-bit microprocessor called the 8086. This processor was a major improvement over the previous generation 8080/8085 series Intel microprocessors in several ways. First, the 8086's capacity of 1 megabyte of memory exceeded the 8080/8085's capability of handling a maximum of 64K bytes of memory. Second, the 8080/8085 was an 8-bit system, meaning that the microprocessor could work on only 8 bits of data at a time. Data larger than 8 bits had to be broken into 8-bit pieces to be processed by the CPU. In contrast, the 8086 is a 16-bit microprocessor. Third, the 8086 was a pipelined processor, as opposed to the nonpipelined 8080/8085. In a system with pipelining, the data and address buses are busy transferring data while the CPU is processing information, thereby increasing the effective processing power of the microprocessor. Although pipelining was a common feature of mini- and mainframe computers, Intel was a pioneer in putting pipelining on a single-chip microprocessor. Section 1.2 discusses pipelining. Table 1-1 shows the evolution of Intel microprocessors up to the 8088.

Table 1-1: Evolution of Intel's Microprocessors (from the 8008 to the 8088)

Product	8008	8080	8085	8086	8088
Year introduced	1972	1974	1976	1978	1979
Technology	PMOS	NMOS	NMOS	NMOS	NMOS
Number of pins	18	40	40	40	40
Number of transistors	3000	4500	6500	29,000	29,000
Number of instructions	66	111	113	133	133
Physical memory	16K	64K	64K	1M	1M
Virtual memory	None	None	None	None	None
Internal data bus	8	8	8	16	16
External data bus	8	8	8	16	8
Address bus	8	16	16	20	20
Data types	8	8	8	8/16	8/16

Evolution from 8086 to 8088

The 8086 is a microprocessor with a 16-bit data bus internally and externally, meaning that all registers are 16 bits wide and there is a 16-bit data bus to transfer data in and out of the CPU. Although the introduction of the 8086 marked a great advancement over the previous generation of microprocessors, there was still some resistance in using the 16-bit external data bus since at that time all peripherals were designed around an 8-bit microprocessor. In addition, a printed circuit board with a 16-bit data bus was much more expensive. Therefore, Intel came out with the 8088 version. It is identical to the 8086 as far as programming is concerned, but externally it has an 8-bit data bus instead of a 16-bit bus. It has the same memory capacity, 1 megabyte.

Success of the 8088

In 1981, Intel's fortunes changed forever when IBM picked up the 8088 as their microprocessor of choice in designing the IBM PC. The 8088-based IBM PC was an enormous success, largely because IBM and Microsoft (the developer of the MS-DOS/Windows operating system) made it an open system, meaning that all documentation and specifications of the hardware and software of the PC were made public. This made it possible for many other vendors to clone the hardware successfully and thus spawned a major growth in both hardware and software designs based on the IBM PC. This is in contrast with the Apple computer, which was a closed system, blocking any attempt at cloning by other manufacturers, both domestically and overseas.

Other microprocessors: the 80286, 80386, and 80486

With a major victory behind Intel and a need from PC users for a more powerful microprocessor, Intel introduced the 80286 in 1982. Its features included 16-bit internal and external data buses; 24 address lines, which give 16 megabytes of memory ($2^{24} = 16$ megabytes); and most significantly, virtual memory. The 80286 can operate in one of two modes: real mode or protected mode. Real mode is simply a faster 8088/8086 with the same maximum of 1 megabyte of memory. Protected mode allows for 16M of memory but is also capable of protecting the operating system and programs from accidental or deliberate destruction by a user, a feature that is absent in the single-user 8088/8086. Virtual memory is a way of fooling the microprocessor into thinking that it has access to an almost unlimited amount of memory by swapping data between disk storage and RAM. IBM picked up the 80286 for the design of the IBM PC AT, and the clone makers followed IBM's lead.

With users demanding even more powerful systems, in 1985 Intel introduced the 80386 (sometimes called 80386DX), internally and externally a 32-bit microprocessor with a 32-bit address bus. It is capable of handling physical memory of up to 4 gigabytes (2^{32}). Virtual memory was increased to 64 terabytes (2^{46}). All microprocessors discussed so far were general-purpose microprocessors and could not handle mathematical calculations rapidly. For this reason, Intel introduced numeric data processing chips, called math coprocessors, such as the 8087, 80287, and 80387. Later Intel introduced the 386SX, which is internally identical to the 80386 but has a 16-bit external data bus and a 24-bit address bus, which gives a capacity of 16 megabytes (2^{24}) of memory. This makes the 386SX system much cheaper. With the introduction of the 80486 in 1989, Intel put a greatly enhanced version of the 80386 and the math coprocessor on a single chip plus additional features such as cache memory. Cache memory is static RAM with a very fast access time. Table 1-2 summarizes the evolution of Intel's microprocessors from the 8086 to the Pentium Pro. It must be noted that all programs written for the 8088/86 will run on 286, 386, and 486 computers.

Table 1-2: Evolution of Intel's Microprocessors (from the 8086 to the Pentium Pro)

Product	8086	80286	80386	80486	Pentium	Pentium Pro
Year Introduced	1978	1982	1985	1989	1993	1995
Technology	NMOS	NMOS	CMOS	CMOS	BICMOS	BICMOS
Clock rate (MHz)	3–10	10–16	16–33	25–33	60, 66	150
Number of pins	40	68	132	168	273	387
Number of transistors	29,000	134,000	275,000	1.2 mill.	3.1 mill.	5.5 mill.
Physical memory	1M	16M	4G	4G	4G	64G
Virtual memory	None	1G	64T	64T	64T	64T
Internal data bus	16	16	32	32	32	32
External data bus	16	16	32	32	64	64
Address bus	20	24	32	32	32	36
Data types	8/16	8/16	8/16/32	8/16/32	8/16/32	8/16/32

e-source

See Intel's Microprocessor Quick Reference Guide at:

http://www.intel.com/pressroom/kits/quickreffam.htm

Table 1-3: Evolution of Intel x86 Microprocessors: From Pentium II to Itanium

Product	Pentium II	Pentium III	Pentium 4	Itanium II
Year introduced	1997	1999	2000	2002
Technology	BICMOS	BICMOS	BICMOS	BICMOS
Number of transistors	7.5 mill.	9.5 mill.	42 mill.	220 mill.
Cache size	512K	512K	512K	3MB
Physical memory	64G	64G	64G	64G
Virtual memory	64T	64T	64T	64T
Internal data bus	32	32	32	64
External data bus	64	64	64	64
Address bus	36	36	36	64
Data types	8/16/32	8/16/32	8/16/32	8/16/32/64

Pentium and Pentium Pro

In 1992, Intel announced release of the newest x86 microprocessor, the Intel Pentium. It was given the name Pentium instead of the expected name 80586 because numbers cannot be copyrighted, whereas a name such as Pentium can be copyrighted. By using submicron fabrication technology, Intel designers were able to utilize more than 3 million transistors on the Pentium chip. Upon its release, the Pentium had speeds of 60 and 66 MHz, but new design features made its processing speed twice that of the 66-MHz 80486 and over 300 times faster than that of the original 8088. The Pentium processor is fully compatible with previous x86 processors but includes several new features, including separate 8K cache memory for code and data, a 64-bit bus, and a vastly improved floating-point processor. The Pentium is packaged in a 273-pin PGA chip. It uses BICMOS technology, which combines the speed of bipolar transistors with the power efficiency of CMOS technology. Although it has a 64-bit data bus, its registers are 32-bit and it has a 32-bit address bus capable of addressing 4 gigabytes of memory. In 1995 Intel introduced the Pentium Pro, the sixth generation of the x86 family. It is an enhanced version of the Pentium that uses 5.5 million transistors. It was designed to be used primarily for 32-bit servers and workstations. Table 1-3 shows the evolution of Intel's microprocessors from the Pentium II to the Itanium II.

Pentium II

In 1997 Intel introduced its Pentium II processor. This 7.5-million-transistor processor featured MMX (MultiMedia extention) technology incorporated into the CPU. MMX allows for fast graphics and audio processing. In 1998 the Pentium II Xeon processor was released. Its primary market is for servers and workstations. In 1999 the Celeron was released. Its lower cost and good performance make it ideal for PCs used to meet educational and home business needs.

Pentium III

In 1999 Intel released the Pentium III. This 9.5-million-transistor processor includes 70 new instructions called SIMD that enhance video and audio performance in such areas as 3-D imaging, and streaming audio that have become common features of online computing. In 1999 Intel also introduced the Pentium III Xeon processor, designed more for servers and business workstations with multiprocessor configurations.

Pentium 4

The Pentium 4, which debuted late in 1999, boasts the speeds of 1.4 to 1.5 GHz. The Pentium 4 represents the first completely new architecture since the development of the Pentium Pro. The new 32-bit architecture, called NetBurst, is designed for heavy multimedia processing such as video, music, and graphic file manipulation on the Internet. The system bus operates at 400 MHz. In addition, new cache and pipelining technology and an expansion of the multimedia instruction set are designed to make the P4 a high-end media processing microprocessor.

Intel 64 Architecture

Intel has selected Itanium as the new brand name for the first product in its 64-bit family of processors, formerly called Merced. The evolution of microprocessors is increasingly influenced by the evolution of the Internet. The Itanium architecture is designed to meet Internet-driven needs for powerful servers and high-performance workstations. The Itanium will have the ability to execute many instructions simultaneously plus extremely large memory capabilities. See Chapter 24 for more.

Review Questions

1. Name three features of the 8086 that were improvements over the 8080/8085.
2. What is the major difference between 8088 and 8086 microprocessors?
3. Give the size of the address bus and physical memory capacity of the following:
   (a) 8086 (b) 80286 (c) 80386
4. The 80286 is a _____-bit microprocessor, whereas the 80386 is a _____-bit microprocessor.
5. State the major difference between the 80386 and the 80386SX.
6. List additional features introduced with the 80286 that were not present in the 8086.
7. List additional features of the 80486 that were not present in the 80386.
8. List additional features of the Pentium that were not present in the 80486.
9. How many transistors did the Pentium II use?
10. Which microprocessor was the first to incorporate MMX technology on-chip?
11. Give the additional features of the Pentium II that were not present in the Pentium.
12. Give all the data types supported by the Pentium 4.
13. Give all the data types supported by the Itanium.
14. True or false. Itanium has a 64-bit architecture.

SECTION 1.2: INSIDE THE 8088/86

In this section we explore concepts important to the internal operation of the 8088/86, such as pipelining and registers. See the block diagram in Figure 1-1.

Pipelining

There are two ways to make the CPU process information faster: increase the working frequency or change the internal architecture of the CPU. The first option is technology dependent, meaning that the designer must use whatever technology is available at the time, with consideration for cost. The technology and materials used in making ICs (integrated circuits) determine the working frequency, power consumption, and the number of transistors packed into a single-chip microprocessor. More discussion of IC technology is given in Chapter 25. It is sufficient for the purpose at hand to say that designers can make the CPU work faster by increasing the frequency under which it runs if technology and cost allow. The second option for improving the processing power of the CPU has to do with the internal working of the CPU. In the 8085 microprocessor, the CPU could either fetch or execute at a given time. In other words, the CPU had to fetch an instruction from memory, then execute it and then fetch again, execute it, and so on.

The idea of pipelining in its simplest form is to allow the CPU to fetch and execute at the same time as shown in Figure 1-2.

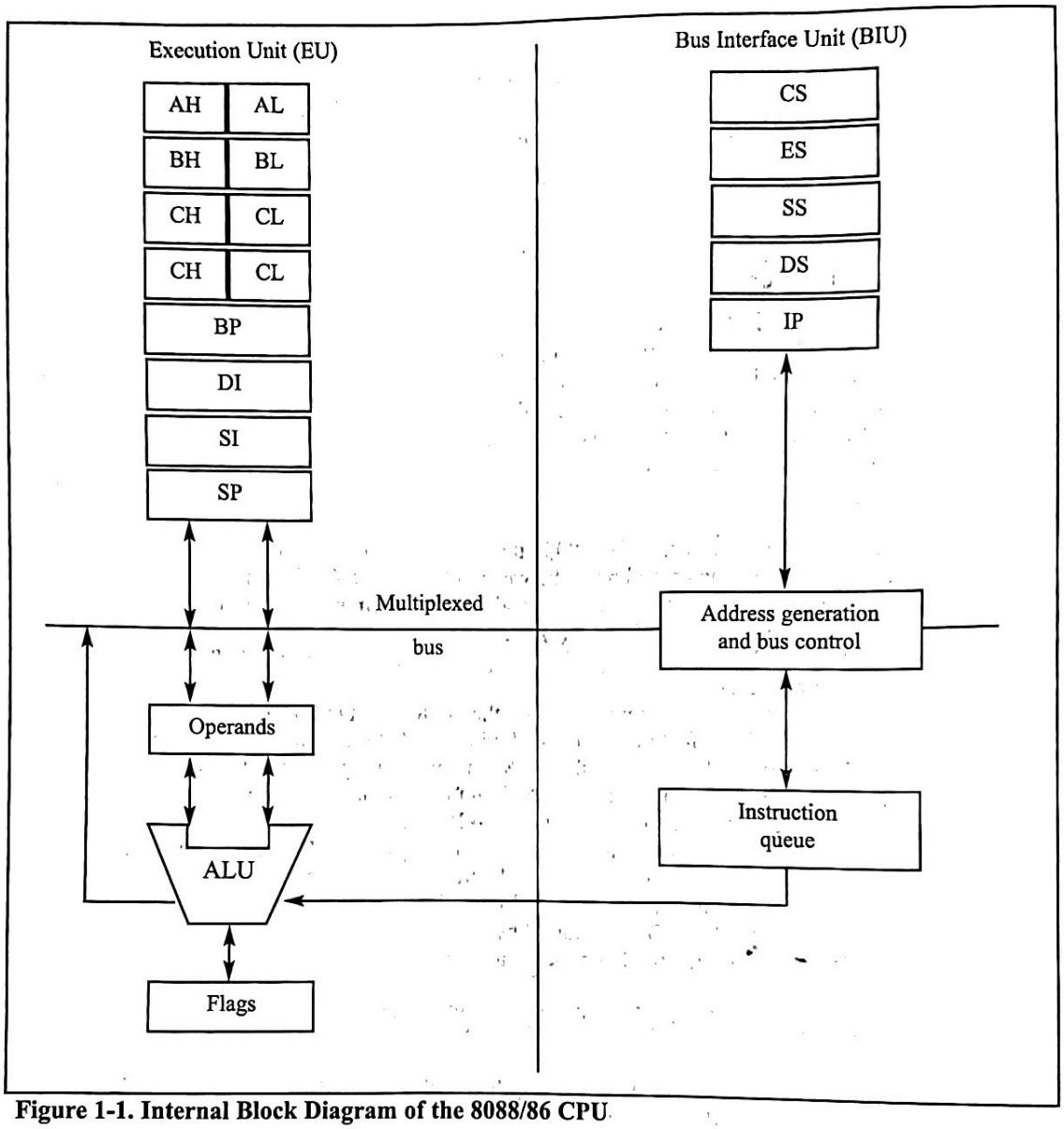


Figure 1-1. Internal Block Diagram of the 8088/86 CPU

(Reprinted by permission of Intel Corporation, Copyright Intel Corp. 1989)

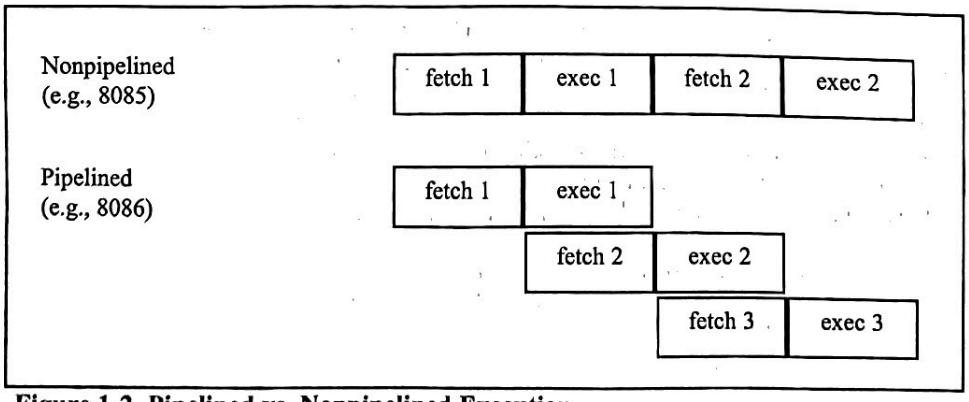
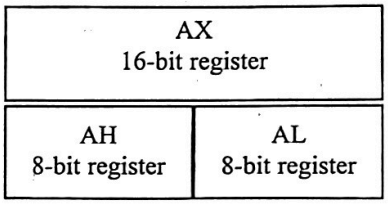


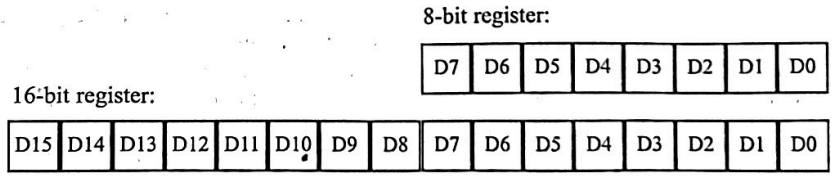
Figure 1-2. Pipelined vs. Nonpipelined Execution

Intel implemented the concept of pipelining in the 8088/86 by splitting the internal structure of the microprocessor into two sections: the execution unit (EU) and the bus interface unit (BIU). These two sections work simultaneously. The BIU accesses memory and peripherals while the EU executes instructions previously fetched. This works only if the BIU keeps ahead of the EU; thus the BIU of the 8088/86 has a buffer, or queue (see Figure 1-1). The buffer is 4 bytes long in the 8088 and 6 bytes in the 8086. If any instruction takes too long to execute, the queue is filled to its maximum capacity and the buses will sit idle. The BIU fetches a new instruction whenever the queue has room for 2 bytes in the 6-byte 8086 queue, and for 1 byte in the 4-byte 8088 queue. In some circumstances, the microprocessor must flush out the queue. For example, when a jump instruction is executed, the BIU starts to fetch information from the new location in memory and information in the queue that was fetched previously is discarded. In this situation the EU must wait until the BIU fetches the new instruction. This is referred to in computer science terminology as a branch penalty. In a pipelined CPU, this means that too much jumping around reduces the efficiency of a program. Pipelining in the 8088/86 has two stages, fetch and execute, but in more powerful computers pipelining can have many stages. The concept of pipelining combined with an increased number of data bus pins has, in recent years, led to the design of very powerful microprocessors.

Registers



In the CPU, registers are used to store information temporarily. That information could be one or two bytes of data to be processed or the address of data. The registers of the 8088/86 fall into the six categories outlined in Table 1-4. The general-purpose registers in 8088/86 microprocessors can be accessed as either 16-bit or 8-bit registers. All other registers can be accessed only as the full 16 bits. In the 8088/86, data types are either 8 or 16 bits. To access 12-bit data, for example, a 16-bit register must be used with the highest 4 bits set to 0. The bits of a register are numbered in descending order, as shown below.



Different registers in the 8088/86 are used for different functions, and since some instructions use only specific registers to perform their tasks, the use of registers will be described in the context of instructions and their application in a given program. The first letter of each general register indicates its use. AX is used for the accumulator, BX as a base addressing register, CX as a counter in loop operations, and DX to point to data in I/O operations. Table 1-4 lists the registers of the 8088/86/286.

Table 1-4: Registers of the 8088/86/286 by Category

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	16	SP (stack pointer), BP (base pointer)
Index	16	SI (source index), DI (destination index)
Segment	16	CS (code segment), DS (data segment), SS (stack segment), ES (extra segment)
Instruction	16	IP (instruction pointer)
Flag	16	FR (flag register)

Note: The general registers can be accessed as the full 16 bits (such as AX), or as the high byte only (AH) or low byte only (AL).

Review Questions

1. Explain the functions of the EU and the BIU.
2. What is pipelining, and how does it make the CPU execute faster?
3. Registers of the 8086 are either _____ bits or _____ bits in length.
4. List the 16-bit registers of the 8086.

SECTION 1.3: INTRODUCTION TO ASSEMBLY PROGRAMMING

While the CPU can work only in binary, it can do so at very high speeds. However, it is quite tedious and slow for humans to deal with 0s and 1s in order to program the computer. A program that consists of 0s and 1s is called *machine language*, and in the early days of the computer, programmers actually coded programs in machine language. Although the hexadecimal system was used as a more efficient way to represent binary numbers, the process of working in machine code was still cumbersome for humans. Eventually, *Assembly languages* were developed, which provided mnemonics for the machine code instructions, plus other features that made programming faster and less prone to error. The term *mnemonic* is typically used in computer science and engineering literature to refer to codes and abbreviations that are relatively easy to remember. Assembly language programs must be translated into machine code by a program called an *assembler*. Assembly language is referred to as a *low-level language* because it deals directly with the internal structure of the CPU. To program in Assembly language, the programmer must know the number of registers and their size, as well as other details of the CPU.

Today, one can use many different programming languages, such as C/C++, BASIC, C#, and numerous others. These languages are called *high-level languages* because the programmer does not have to be concerned with the internal details of the CPU. Whereas an assembler is used to translate an Assembly language program into machine code (sometimes called *object code*), high-level languages are translated into machine code by a program called a *compiler*. For instance, to write a program in C, one must use a C compiler to translate the program into machine language.

There are numerous assemblers available for translating x86 Assembly language programs into machine code. One of the most commonly used assemblers, MASM by Microsoft, is introduced in Chapter 2. The present chapter is designed to correspond to Appendix A: DEBUG Programming. The program in this chapter can be entered and run with the use of the DEBUG program. If you are not familiar with DEBUG, refer to Appendix A for a tutorial introduction. The DEBUG utility is provided with the Microsoft Windows operating system and therefore is widely accessible.

Assembly language programming

An Assembly language program consists of, among other things, a series of lines of Assembly language instructions. An Assembly language instruction consists of a mnemonic, optionally followed by one or two operands. The operands are the data items being manipulated, and the mnemonics are the commands to the CPU, telling it what to do with those items. We introduce Assembly language programming with two widely used instructions: the move and add instructions.

MOV instruction

Simply stated, the MOV instruction copies data from one location to another. It has the following format:

```
MOV  destination,source ;copy source operand to destination
```

This instruction tells the CPU to move (in reality, copy) the source operand to the destination operand. For example, the instruction "`MOV DX,CX`" copies the contents of register CX to register DX. After this instruction is executed, register DX will have the same value as register CX. The MOV instruction does not affect the source operand. The

following program first loads CL with value 55H, then moves this value around to various registers inside the CPU.

```
MOV  CL,55H ;move 55H into register CL
MOV  DL,CL ;copy the contents of CL into DL (now DL=CL=55H)
MOV  AH,DL ;copy the contents of DL into AH (now AH=DL=55H)
MOV  AL,AH ;copy the contents of AH into AL (now AL=AH=55H)
MOV  BH,CL ;copy the contents of CL into BH (now BH=CL=55H)
MOV  CH,BH ;copy the contents of BH into CH (now CH=BH=55H)
```

The use of 16-bit registers is demonstrated below.

```
MOV  CX,468FH  ;move 468FH into CX (now CH=46,CL=8F)
MOV  AX,CX    ;copy contents of CX to AX (now AX=CX=468FH)
MOV  DX,AX    ;copy contents of AX to DX (now DX=AX=468FH)
MOV  BX,DX    ;copy contents of DX to BX (now BX=DX=468FH)
MOV  DI,BX    ;now DI=BX=468FH
MOV  SI,DI    ;now SI=DI=468FH
MOV  DS,SI    ;now DS=SI=468FH
MOV  BP,DI    ;now BP=DI=468FH
```

In the 8086 CPU, data can be moved among all the registers shown in Table 1-4 (except the flag register) as long as the source and destination registers match in size. Code such as "`MOV AL,DX`" will cause an error, since one cannot move the contents of a 16-bit register into an 8-bit register. There is no such instruction as "`MOV FR,AX`". Loading the flag register is done through other means, discussed in later chapters.

If data can be moved among all registers including the segment registers, can data be moved directly into all registers? The answer is no. Data can be moved directly into nonsegment registers only, using the MOV instruction. For example, look at the following instructions to see which are legal and which are illegal.

```
MOV  AX,58FCH  ;move 58FCH into AX   (LEGAL)
MOV  DX,6678H  ;move 6678H into DX   (LEGAL)
MOV  SI,924BH  ;move 924B  into SI   (LEGAL)
MOV  BP,2459H  ;move 2459H into BP   (LEGAL)
MOV  DS,2341H  ;move 2341H into DS   (ILLEGAL)
MOV  CX,8876H  ;move 8876H into CX   (LEGAL)
MOV  CS,3F47H  ;move 3F47H into CS   (ILLEGAL)
MOV  BH,99H    ;move 99H into BH     (LEGAL)
```

From the discussion above, note the following three points:

1. Values cannot be loaded directly into any segment register (CS, DS, ES, or SS). To load a value into a segment register, first load it to a nonsegment register and then move it to the segment register, as shown next.

```
MOV  AX,2345H  ;load 2345H into AX
MOV  DS,AX    ;then load the value of AX into DS

MOV  DI,1400H  ;load 1400H into DI
MOV  ES,DI    ;then move it into ES, now ES=DI=1400
```

2. If a value less than FFH is moved into a 16-bit register, the rest of the bits are assumed to be all zeros. For example, in "`MOV BX,5`" the result will be BX = 0005; that is, BH = 00 and BL = 05.
3. Moving a value that is too large into a register will cause an error.

```
MOV  BL,7F2H      ;ILLEGAL: 7F2H is larger than 8 bits
MOV  AX,2FE456H   ;ILLEGAL: the value is larger than AX
```

ADD instruction

The ADD instruction has the following format:

```
ADD  destination,source  ;ADD the source operand to the destination
```

The ADD instruction tells the CPU to add the source and the destination operands and put the result in the destination. To add two numbers such as 25H and 34H, each can be moved to a register and then added together:

```
MOV  AL,25H  ;move 25 into AL
MOV  BL,34H  ;move 34 into BL
ADD  AL,BL   ;AL = AL + BL
```

Executing the program above results in AL = 59H (25H + 34H = 59H) and BL = 34H. Notice that the contents of BL do not change. The program above can be written in many ways, depending on the registers used. Another way might be:

```
MOV  DH,25H  ;move 25 into DH
MOV  CL,34H  ;move 34 into CL
ADD DH,CL    ;add CL to DH: DH = DH + CL
```

The program above results in DH = 59H and CL = 34H. There are always many ways to write the same program. One question that might come to mind after looking at the program above is whether it is necessary to move both data items into registers before adding them together. No, it is not necessary. Look at the following variation:

```
MOV DH,25H  ;load one operand into DH
ADD DH,34H  ;add the second operand to DH
```

In the case above, while one register contained one value, the second value followed the instruction as an operand. This is called an *immediate operand.* The examples shown so far for the ADD and MOV instructions show that the source operand can be either a register or immediate data. In the examples above, the destination operand has always been a register. The format for Assembly language instructions, descriptions of their use, and a listing of legal operand types are provided in Appendix B.

The largest number that an 8-bit register can hold is FFH. To use numbers larger than FFH (255 decimal), 16-bit registers such as AX, BX, CX, or DX must be used. For example, to add 34EH and 6A5H, the following program can be used:

```
MOV AX,34EH  ;move 34EH into AX
MOV DX,6A5H  ;move 6A5H into DX
ADD  DX,AX   ;add AX to DX: DX = DX + AX
```

Running the program above gives DX = 9F3H (34E + 6A5 = 9F3) and AX = 34E. Again, any 16-bit nonsegment registers could have been used to perform the action above:

```
MOV  CX,34EH  ;load 34EH into CX
ADD CX,6A5H ;add 6A5H to CX (now CX=9F3H)
```

The general-purpose registers are typically used in arithmetic operations. Register AX is sometimes referred to as the *accumulator.*

Review Questions

1. Write the Assembly language instruction to move value 1234H into register BX.
2. Write the Assembly language instructions to add the values 16H and ABH. Place the result in register AX.
3. No value can be moved directly into which registers?
4. What is the largest hex value that can be moved into a 16-bit register? Into an 8-bit register? What are the decimal equivalents of these hex values?

SECTION 1.4: INTRODUCTION TO PROGRAM SEGMENTS

A typical Assembly language program consists of at least three segments: a code segment, a data segment, and a stack segment. The *code segment* contains the Assembly language instructions that perform the tasks that the program was designed to accomplish. The *data segment* is used to store information (data) that needs to be processed by the instructions in the code segment. The *stack* is used by the CPU to store information temporarily. In this section we describe the code and data segments of a program in the context of some examples and discuss the way data is stored in memory. The stack segment is covered in Section 1.5.

Origin and definition of the segment

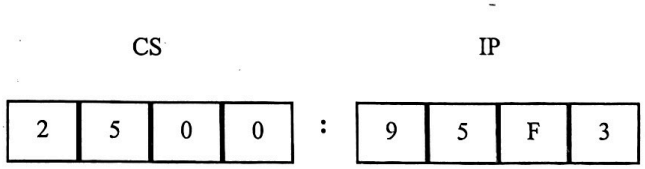
A segment is an area of memory that includes up to 64K bytes and begins on an address evenly divisible by 16 (such an address ends in 0H). The segment size of 64K bytes came about because the 8085 microprocessor could address a maximum of 64K bytes of physical memory since it had only 16 pins for the address lines (2^{16} = 64K). This limitation was carried into the design of the 8088/86 to ensure compatibility. Whereas in the 8085 there was only 64K bytes of memory for all code, data, and stack information, in the 8088/86 there can be up to 64K bytes of memory assigned to each category. Within an Assembly language program, these categories are called the *code segment*, *data segment*, and *stack segment*. For this reason, the 8088/86 can only handle a maximum of 64K bytes of code, 64K bytes of data, and 64K bytes of stack at any given time, although it has a range of 1 megabyte of memory because of its 20 address pins (2^{20} = 1 megabyte). How to move this window of 64K bytes to cover all 1 megabyte of memory is discussed below, after we discuss logical address and physical address.

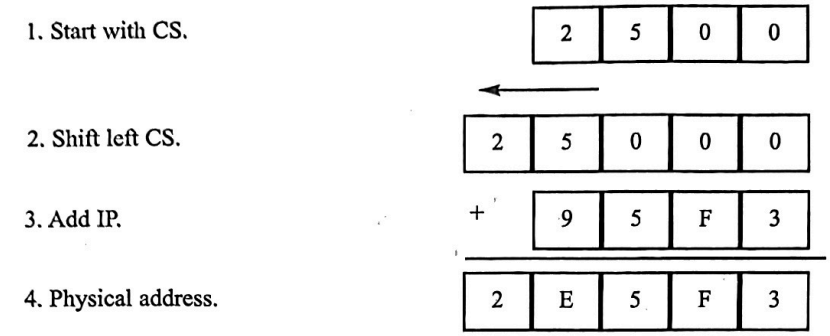
Logical address and physical address

In Intel literature concerning the 8086, there are three types of addresses mentioned frequently: the physical address, the offset address, and the logical address. The *physical address* is the 20-bit address that is actually put on the address pins of the 8086 microprocessor and decoded by the memory interfacing circuitry. This address can have a range of 00000H to FFFFFH for the 8086 and real-mode 286, 386, and 486 CPUs. This is an actual physical location in RAM or ROM within the 1 megabyte memory range. The *offset address* is a location within a 64K-byte segment range. Therefore, an offset address can range from 0000H to FFFFH. The *logical address* consists of a segment value and an offset address. The differences among these addresses and the process of converting from one to another is best understood in the context of some examples, as shown next.

Code segment

To execute a program, the 8086 fetches the instructions (opcodes and operands) from the code segment. The logical address of an instruction always consists of a CS (code segment) and an IP (instruction pointer), shown in CS:IP format. The physical address for the location of the instruction is generated by shifting the CS left one hex digit and then adding it to the IP. IP contains the offset address. The resulting 20-bit address is called the physical address since it is put on the external physical address bus pins to be decoded by the memory decoding circuitry. To clarify this important concept, assume values in CS and IP as shown in the diagram. The offset address is contained in IP; in this case it is 95F3H. The logical address is CS:IP, or 2500:95F3H. The physical address will be 25000 + 95F3 = 2E5F3H. The physical address of an instruction can be calculated as follows:





The microprocessor will retrieve the instruction from memory locations starting at 2E5F3. Since IP can have a minimum value of 0000H and a maximum of FFFFH, the logical address range in this example is 2500:0000 to 2500:FFFF. This means that the lowest memory location of the code segment above will be 25000H (25000 + 0000) and the highest memory location will be 34FFFH (25000 + FFFF). What happens if the desired instructions are located beyond these two limits? The answer is that the value of CS must be changed to access those instructions. See Example 1-1.

Example 1-1

If CS = 24F6H and IP = 634AH, show (a) the logical address, and (b) the offset address. Calculate (c) the physical address, (d) the lower range, and (e) the upper range of the code segment.

Solution:

(a) 24F6:634A (b) 634A (c) 2B2AA (24F60 + 634A)
(d) 24F60 (24F60 + 0000) (e) 34F5F (24F60 + FFFF)

Logical address vs. physical address in the code segment

In the code segment, CS and IP hold the logical address of the instructions to be executed. The following Assembly language instructions have been assembled (translated into machine code) and stored in memory. The three columns show the logical address of CS:IP, the machine code stored at that address, and the corresponding Assembly language code.

LOGICAL ADDRESS CS:IP	MACHINE LANGUAGE OPCODE AND OPERAND	ASSEMBLY LANGUAGE MNEMONICS AND OPERAND
1132:0100	B057	MOV AL,57
1132:0102	B686	MOV DH,86
1132:0104	B272	MOV DL,72
1132:0106	89D1	MOV CX,DX
1132:0108	88C7	MOV BH,AL
1132:010A	B39F	MOV BL,9F
1132:010C	B420	MOV AH,20
1132:010E	01D0	ADD AX,DX
1132:0110	01D9	ADD CX,BX
1132:0112	05351F	ADD AX,1F35

The program above shows that the byte at address 1132:0100 contains B0, which is the opcode for moving a value into register AL, and address 1132:0101 contains the operand (in this case 57) to be moved to AL. Therefore, the instruction "MOV AL, 57" has a machine code of B057, where B0 is the opcode and 57 is the operand. Similarly, the

machine code B686 is located in memory locations 1132:0102 and 1132:0103 and represents the opcode and the operand for the instruction "MOV DH,86". The physical address is an actual location within RAM (or even ROM). The following are the physical addresses and the contents of each location for the program above. Remember that it is the physical address that is put on the address bus by the 8086 CPU to be decoded by the memory circuitry:

LOGICAL ADDRESS	PHYSICAL ADDRESS	MACHINE CODE CONTENTS
1132:0100	11420	B0
1132:0101	11421	57
1132:0102	11422	B6
1132:0103	11423	86
1132:0104	11424	B2
1132:0105	11425	72
1132:0106	11426	89
1132:0107	11427	D1
1132:0108	11428	88
1132:0109	11429	C7
1132:010A	1142A	B3
1132:010B	1142B	9F
1132:010C	1142C	B4
1132:010D	1142D	20
1132:010E	1142E	01
1132:010F	1142F	D0
1132:0110	11430	01
1132:0111	11431	D9
1132:0112	11432	05
1132:0113	11433	35
1132:0114	11434	1F

Data segment

Assume that a program is being written to add 5 bytes of data, such as 25H, 12H, 15H, 1FH, and 2BH, where each byte represents a person's daily overtime pay. One way to add them is as follows:

```
MOV  AL,00H  ;initialize AL
ADD  AL,25H  ;add 25H to AL
ADD  AL,12H  ;add 12H to AL
ADD  AL,15H  ;add 15H to AL
ADD  AL,1FH  ;add 1FH to AL
ADD  AL,2BH  ;add 2BH to AL
```

In the program above, the data and code are mixed together in the instructions. The problem with writing the program this way is that if the data changes, the code must be searched for every place the data is included, and the data retyped. For this reason, the idea arose to set aside an area of memory strictly for data. In x86 microprocessors, the area of memory set aside for data is called the *data segment*. Just as the code segment is associated with CS and IP as its segment register and offset, the data segment uses register DS and an offset value.

The following demonstrates how data can be stored in the data segment and the program rewritten so that it can be used for any set of data. Assume that the offset for the data segment begins at 200H. The data is placed in memory locations:

```
DS:0200 = 25
DS:0201 = 12
DS:0202 = 15
DS:0203 = 1F
DS:0204 = 2B
```

and the program can be rewritten as follows:

```
MOV   AL,0          ;clear AL
ADD   AL,[ 0200]    ;add the contents of DS:200 to AL
ADD   AL,[ 0201]    ;add the contents of DS:201 to AL
ADD   AL,[ 0202]    ;add the contents of DS:202 to AL
ADD   AL,[ 0203]    ;add the contents of DS:203 to AL
ADD   AL,[ 0204]    ;add the contents of DS:204 to AL
```

Notice that the offset address is enclosed in brackets. The brackets indicate that the operand represents the address of the data and not the data itself. If the brackets were not included, as in "`MOV AL,0200`", the CPU would attempt to move 200 into AL instead of the contents of offset address 200. Keep in mind that there is one important difference in the format of code for MASM and DEBUG in that DEBUG assumes that all numbers are in hex (no "H" suffix is required), whereas MASM assumes that they are in decimal and the "H" must be included for hex data.

This program will run with any set of data. Changing the data has no effect on the code. Although this program is an improvement over the preceding one, it can be improved even further. If the data had to be stored at a different offset address, say 450H, the program would have to be rewritten. One way to solve this problem would be to use a register to hold the offset address, and before each ADD, to increment the register to access the next byte. Next a decision must be made as to which register to use. The 8088/86 allows only the use of registers BX, SI, and DI as offset registers for the data segment. In other words, while CS uses only the IP register as an offset, DS uses only BX, DI, and SI to hold the offset address of the data. The term *pointer* is often used for a register holding an offset address. In the following example, BX is used as a pointer:

```
MOV   AL,0          ;initialize AL
MOV   BX,0200H      ;BX points to offset addr of first byte
ADD   AL,[ BX]      ;add the first byte to AL
INC   BX            ;increment BX to point to the next byte
ADD   AL,[ BX]      ;add the next byte to AL
INC   BX            ;increment the pointer
ADD   AL,[ BX]      ;add the next byte to AL
INC   BX            ;increment the pointer
ADD   AL,[ BX]      ;add the last byte to AL
```

The INC instruction adds 1 to (increments) its operand. "`INC BX`" achieves the same result as "`ADD BX,1`". For the program above, if the offset address where data is located is changed, only one instruction will need to be modified and the rest of the program will be unaffected. Examining the program above shows that there is a pattern of two instructions being repeated. This leads to the idea of using a loop to repeat certain instructions. Implementing a loop requires familiarity with the flag register, discussed later in this chapter.

Logical address and physical address in the data segment

The physical address for data is calculated using the same rules as for the code segment. That is, the physical address of data is calculated by shifting DS left one hex digit and adding the offset value, as shown in Examples 1-2, 1-3, and 1-4.

Little endian convention

Previous examples used 8-bit or 1-byte data. In this case the bytes are stored one after another in memory. What happens when 16-bit data is used? For example:

```
MOV   AX,35F3H ;load 35F3H into AX
MOV   [ 1500],AX  ;copy the contents of AX to offset 1500H
```

In cases like this, the low byte goes to the low memory location and the high byte

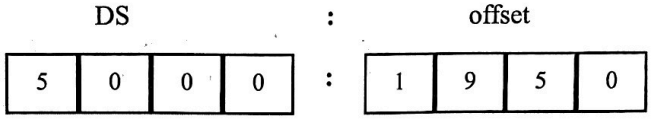
goes to the high memory address. In the example above, memory location DS:1500 contains F3H and memory location DS:1501 contains 35H (DS:1500 = F3 DS:1501 = 35).

This convention is called little endian versus big endian. The origin of the terms *big endian* and *little endian* is from a *Gulliver's Travels* story about how an egg should be opened: from the little end or the big end. In the big endian method, the high byte goes to the low address, whereas in the little endian method, the high byte goes to the high address and the low byte to the low address. See Example 1-5. All Intel microprocessors and many microcontrollers use the little endian convention. Freescale (formerly Motorola) microprocessors, along with some other microcontrollers, use big endian. This difference might seem as trivial as whether to break an egg from the big end or little end, but it is a nuisance in converting software from one camp to be run on a computer of the other camp.

Example 1-2

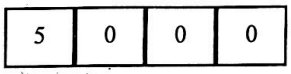
Assume that DS is 5000 and the offset is 1950. Calculate the physical address.

Solution:

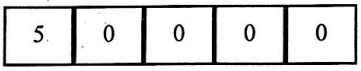


The physical address will be 50000 + 1950 = 51950.

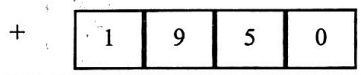
1. Start with DS.



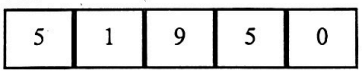
2. Shift DS left.



3. Add the offset.



4. Physical address.



Example 1-3

If DS = 7FA2H and the offset is 438EH, calculate (a) the physical address, (b) the lower range, and (c) the upper range of the data segment. Show (d) the logical address.

Solution:

(a) 83DAE (7FA20 + 438E)
(b) 7FA20 (7FA20 + 0000)
(c) 8FA1F (7FA20 + FFFF)
(d) 7FA2:438E

Example 1-4

Assume that the DS register is 578C. To access a given byte of data at physical memory location 67F66, does the data segment cover the range where the data resides? If not, what changes need to be made?

Solution:

No, since the range is 578C0 to 678BF, location 67F66 is not included in this range. To access that byte, DS must be changed so that its range will include that byte.

Example 1-5

Assume memory locations with the following contents: DS:6826 = 48 and DS:6827 = 22. Show the contents of register BX in the instruction "MOV BX,[6826]".

Solution:
According to the little endian convention used in all x86 microprocessors, register BL should contain the value from the low offset address 6826 and register BH the value from the offset address 6827, giving BL = 48H and BH = 22H.

DS:6826 = 48
DS:6827 = 22

BH	BL
22	48

Extra segment (ES)

ES is a segment register used as an extra data segment. Although in many normal programs this segment is not used, its use is absolutely essential for string operations and is discussed in detail in Chapter 6.

Memory map of the IBM PC

For a program to be executed on the PC, Windows must first load it into RAM. Where in RAM will it be loaded? To answer that question, we must first explain some very important concepts concerning memory in the PC. The 20-bit address of the 8088/86 allows a total of 1 megabyte (1024K bytes) of memory space with the address range 00000–FFFFF. During the design phase of the first IBM PC, engineers had to decide on the allocation of the 1-megabyte memory space to various sections of the PC. This memory allocation is called a *memory map*. The memory map of the IBM PC is shown in Figure 1-3. Of this 1 megabyte, 640K bytes from addresses 00000–9FFFFH were set aside for RAM. The 128K bytes from A0000H to BFFFFH were allocated for video memory. The remaining 256K bytes from C0000H to FFFFFH were set aside for ROM.

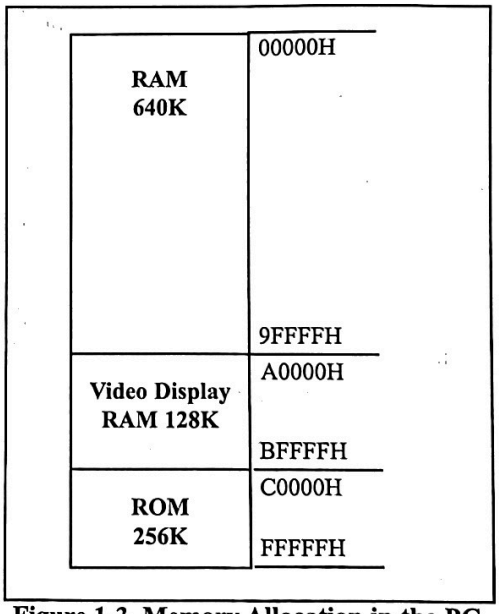


Figure 1-3. Memory Allocation in the PC

More about RAM

In the early 1980s, most PCs came with only 64K to 256K bytes of RAM memory, which was considered more than adequate at the time. Users had to buy memory expansion boards to expand memory up to 640K if they needed additional memory. The need for expansion depends on the Windows version being used and the memory needs of the application software being run. The Windows operating system first allocates the available RAM on the PC for its own use and then lets the rest be used for applications such as word processors. The complicated task of managing RAM memory is left to Windows since the amount of memory used by Windows varies among its various versions and since different computers have different amounts of RAM, plus the fact that the memory needs of application packages vary. For this reason we do not assign any values for the CS, DS, and SS registers since such an assignment means specifying an exact

physical address in the range 00000–9FFFFH, and this is beyond the knowledge of the user. Another reason is that assigning a physical address might work on a given PC but it might not work on a PC with a different OS version and RAM size. In other words, the program would not be portable to another PC. Therefore, memory management is one of the most important functions of the operating system and should be left to Windows. This is very important to remember because in many examples in this book we have values for the segment registers CS, DS, and SS that will be different from the values that readers will get on their PCs. Do not try to assign the value to the segment registers to comply with the values in this book.

Video RAM

From A0000H to BFFFFH is set aside for video. The amount used and the location vary depending on the video board installed on the PC. Table E-2 of Appendix E lists the starting addresses for video boards.

More about ROM

From C0000H to FFFFFH is set aside for ROM. Not all the memory space in this range is used by the PC's ROM. Of this 256K bytes, only the 64K bytes from location F0000H–FFFFFH are used by BIOS (basic input/output system) ROM. Some of the remaining space is used by various adapter cards (such as the network card), and the rest is free. In recent years, newer versions of Windows have gained some very powerful memory management capabilities and can put to good use all the unused memory space beyond 640. The 640K-byte memory space from 00000 to 9FFFFH is referred to as *conventional memory*, while the 384K bytes from A0000H to FFFFFH are called the UMB (*upper memory block*) in Microsoft literature.

Function of BIOS ROM

Since the CPU can only execute programs that are stored in memory, there must be some permanent (nonvolatile) memory to hold the programs telling the CPU what to do when the power is turned on. This collection of programs held by ROM is referred to as BIOS in the PC literature. BIOS, which stands for *basic input-output system*, contains programs to test RAM and other components connected to the CPU. It also contains programs that allow Windows to communicate with peripheral devices such as the keyboard, video, printer, and disk. It is the function of BIOS to test all the devices connected to the PC when the computer is turned on and to report any errors. For example, if the keyboard is disconnected from the PC before the computer is turned on, BIOS will report an error on the screen, indicating that condition. It is only after testing and setting up the peripherals that BIOS will load Windows from disk into RAM and hand over control of the PC to Windows. Although there are occasions when either Windows or applications programs need to use programs in BIOS ROM, Windows always controls the PC once it is loaded.

Review Questions

1. A segment is an area of memory that includes up to ____ bytes.
2. How large is a segment in the 8086? Can the physical address 346E0 be the starting address for a segment? Why or why not?
3. State the difference between the physical and logical addresses.
4. A physical address is a ____-bit address; an offset address is a ____-bit address.
5. Which register is used as the offset register with segment register CS?
6. If BX = 1234H and the instruction "MOV [2400],BX" were executed, what would be the contents of memory locations at offsets 2400 and 2401?

SECTION 1.5: THE STACK

In this section we examine the concept of the stack, its use in x86 microprocessors, and its implementation in the stack segment. Then more advanced concepts relating to segments are discussed, such as overlapping segments.

What is a stack, and why is it needed?

The *stack* is a section of read/write memory (RAM) used by the CPU to store information temporarily. The CPU needs this storage area since there are only a limited number of registers. There must be some place for the CPU to store information safely and temporarily. Now one might ask, why not design a CPU with more registers? The reason is that in the design of the CPU, every transistor is precious and not enough of them are available to build hundreds of registers. In addition, how many registers should a CPU have to satisfy every possible program and application? All applications and programming techniques are not the same. In a similar manner, it would be too costly in terms of real estate and construction costs to build a 50-room house to hold everything one might possibly buy throughout his or her lifetime. Instead, one builds or rents a shed for storage.

Having looked at the advantages of having a stack, what are the disadvantages? The main disadvantage of the stack is its access time. Since the stack is in RAM, it takes much longer to access compared to the access time of registers. After all, the registers are inside the CPU and RAM is outside. This is the reason that some very powerful (and consequently, expensive) computers do not have a stack; the CPU has a large number of registers to work with.

How stacks are accessed

If the stack is a section of RAM, there must be registers inside the CPU to point to it. The two main registers used to access the stack are the SS (stack segment) register and the SP (stack pointer) register. These registers must be loaded before any instructions accessing the stack are used. Every register inside the x86 (except segment registers and SP) can be stored in the stack and brought back into the CPU from the stack memory. The storing of a CPU register in the stack is called a *push*, and loading the contents of the stack into the CPU register is called a *pop*. In other words, a register is pushed onto the stack to store it and popped off the stack to retrieve it. The job of the SP is very critical when push and pop are performed. In the x86, the stack pointer register (SP) points at the current memory location used for the top of the stack and as data is pushed onto the stack it is decremented. It is incremented as data is popped off the stack into the CPU. When an instruction pushes or pops a general-purpose register, it must be the entire 16-bit register. In other words, one must code "`PUSH AX`"; there are no instructions such as "`PUSH AL`" or "`PUSH AH`". The reason that the SP is decremented after the push is to make sure that the stack is growing downward from upper addresses to lower addresses. This is the opposite of the IP (instruction pointer). As was seen in the preceding section, the IP points to the next instruction to be executed and is incremented as each instruction is executed. To ensure that the code section and stack section of the program never write over each other, they are located at opposite ends of the RAM memory set aside for the program and they grow toward each other but must not meet. If they meet, the program will crash. To see how the stack grows, look at the following examples.

Pushing onto the stack

Notice in Example 1-6 that as each PUSH is executed, the contents of the register are saved on the stack and SP is decremented by 2. For every byte of data saved on the stack, SP is decremented once, and since push is saving the contents of a 16-bit register, it is decremented twice. Notice also how the data is stored on the stack. In the x86, the lower byte is always stored in the memory location with the lower address. That is the reason that 24H, the contents of AH, is saved in the memory location with the address 1235 and AL in location 1234.

Popping the stack

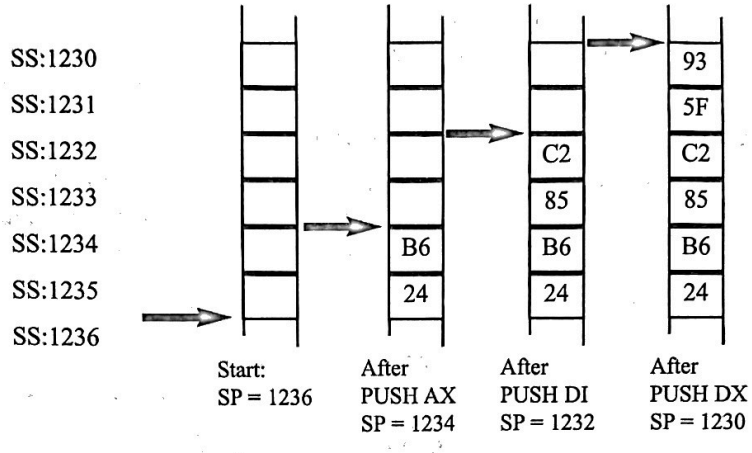
Popping the contents of the stack back into the x86 CPU is the opposite process of pushing. With every pop, the top 2 bytes of the stack are copied to the register specified by the instruction and the stack pointer is incremented twice. Although the data actually remains in memory, it is not accessible since the stack pointer is beyond that point. Example 1-7 demonstrates the POP instruction.

Example 1-6

Assuming that SP = 1236, AX = 24B6, DI = 85C2, and DX = 5F93, show the contents of the stack as each of the following instructions is executed.

```
PUSH  AX
PUSH  DI
PUSH  DX
```

Solution:

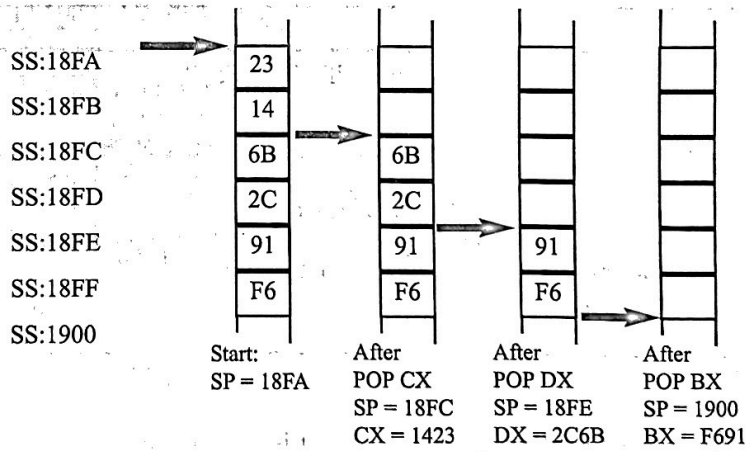


Example 1-7

Assuming that the stack is as shown below, and SP = 18FA, show the contents of the stack and registers as each of the following instructions is executed:

```
POP   CX
POP   DX
POP   BX
```

Solution:



Logical address vs. physical address for the stack

What is the exact physical location of the stack? That depends on the value of the stack segment (SS) register and SP, the stack pointer. To compute physical addresses for the stack, the same principle is applied as was used for the code and data segments. We shift left SS and then add offset SP, the stack pointer register. See Example 1-8.

What values are assigned to the SP and SS, and who assigns them? It is the job of the Windows operating system to assign the values for the SP and SS since memory management is the responsibility of the operating system. Before leaving the discussion of the stack, two points must be made. First, in the x86 literature, the top of the stack is the last stack location occupied. This is different from other CPUs. Second, BP is another register that can be used as an offset into the stack, but it has very special applications and is widely used to access parameters passed between Assembly language programs and high-level language programs such as C.

Example 1-8

If SS = 3500H and the SP is FFFEH,
(a) Calculate the physical address of the stack.
(b) Calculate the lower range.
(c) Calculate the upper range of the stack segment.
(d) Show the stack's logical address.

Solution:
(a) 44FFE (35000 + FFFE)
(b) 35000 (35000 + 0000)
(c) 44FFF (35000 + FFFF)
(d) 3500:FFFE

A few more words about segments in the x86

Can a single physical address belong to many different logical addresses? Yes, look at the case of a physical address value of 15020H. There are many possible logical addresses that represent this single physical address:

Logical address (hex)	Physical address (hex)
1000:5020	15020
1500:0020	15020
1502:0000	15020
1400:1020	15020
1302:2000	15020

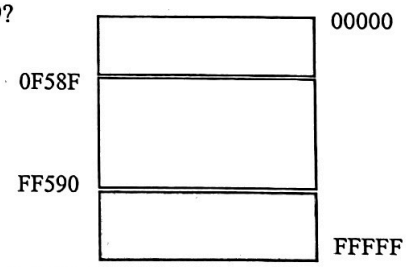
This shows the dynamic behavior of the segment and offset concept in the 8086 CPU. One last point that must be clarified is the case when adding the offset to the shifted segment register results in an address beyond the maximum allowed range of FFFFFH. In that situation, wrap-around will occur. This is shown in Example 1-9.

Example 1-9

What is the range of physical addresses if CS = FF59?

Solution:

The low range is FF590 (FF590 + 0000).
The range goes to FFFFF and wraps around, from 00000 to 0F58F (FF590 + FFFF = 0F58F), as shown in the illustration.



Overlapping

In calculating the physical address, it is possible that two segments can overlap, which is desirable in some circumstances. Figure 1-4 illustrates overlapping and nonoverlapping segments.

Review Questions

1. Which registers are used to access the stack?
2. With each PUSH instruction, the stack pointer register SP is (circle one) incremented/decremented by 2.
3. With each POP instruction, SP is (circle one) incremented/decremented by 2.
4. List three possible logical addresses corresponding to physical address 143F0.

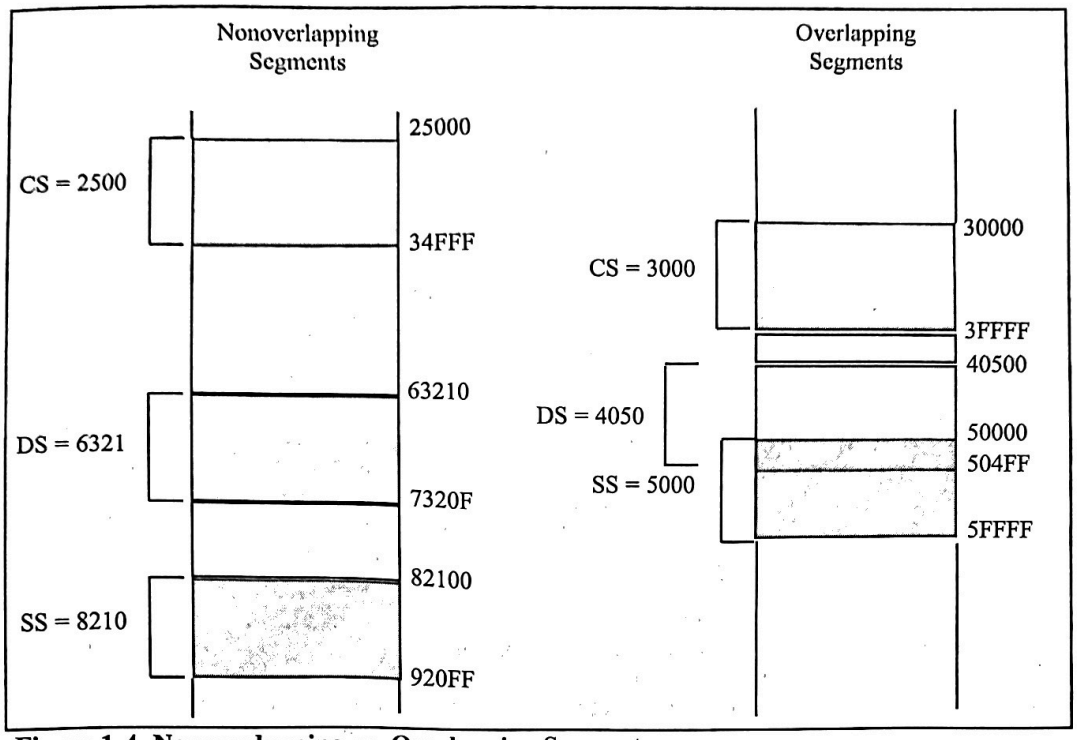


Figure 1-4. Nonoverlapping vs. Overlapping Segments

SECTION 1.6: FLAG REGISTER

In this section we describe the flag register. Many Assembly language instructions alter bits of the flag register and some instructions will function differently based on the information in the flag register. After describing the bits of the flag register and their function, programming examples are given to demonstrate the use of the flag register.

The flag register is a 16-bit register sometimes referred to as the *status register*. Although the register is 16 bits wide, only some of the bits are used. The rest are either undefined or reserved by Intel. Six of the flags are called *conditional flags*, meaning that they indicate some condition that resulted after an instruction was executed. These six are CF, PF, AF, ZF, SF, and OF. The three remaining flags are sometimes called *control flags* since they are used to control the operation of instructions before they are executed. A diagram of the flag register is shown in Figure 1-5.

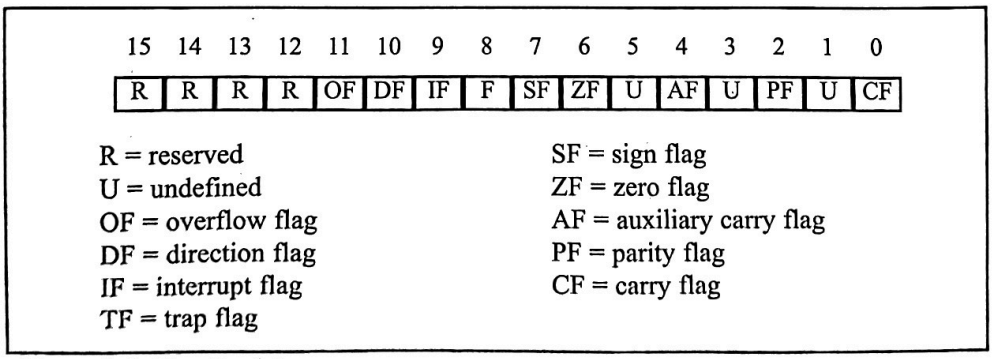


Figure 1-5. Flag Register

Bits of the flag register

Below are listed the bits of the flag register that are used in x86 Assembly language programming. A brief explanation of each bit is given. How these flag bits are used will be seen in programming examples throughout the textbook.

CF, the Carry Flag. This flag is set whenever there is a carry out, either from d7 after an 8-bit operation, or from d15 after a 16-bit data operation.

PF, the Parity Flag. After certain operations, the parity of the result's low-order byte is checked. If the byte has an even number of 1s, the parity flag is set to 1; otherwise, it is cleared.

AF, Auxiliary Carry Flag. If there is a carry from d3 to d4 of an operation, this bit is set; otherwise, it is cleared (set equal to zero). This flag is used by the instructions that perform BCD (binary coded decimal) arithmetic.

ZF, the Zero Flag. The zero flag is set to 1 if the result of an arithmetic or logical operation is zero; otherwise, it is cleared.

SF, the Sign Flag. Binary representation of signed numbers uses the most significant bit as the sign bit. After arithmetic or logic operations, the status of this sign bit is copied into the SF, thereby indicating the sign of the result.

TF, the Trap Flag. When this flag is set it allows the program to single-step, meaning to execute one instruction at a time. Single-stepping is used for debugging purposes.

IF, Interrupt Enable Flag. This bit is set or cleared to enable or disable only the external maskable interrupt requests.

DF, the Direction Flag. This bit is used to control the direction of string operations, which are described in Chapter 6.

OF, the Overflow Flag. This flag is set whenever the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit. In general, the carry flag is used to detect errors in unsigned arithmetic operations. The overflow flag is only used to detect errors in signed arithmetic operations.

Example 1-10

Show how the flag register is affected by the addition of 38H and 2FH.

Solution:

```
        MOV  BH,38H          ;BH= 38H
        ADD  BH,2FH          ;add 2F to BH, now BH=67H

     38      0011  1000
+    2F      0010  1111
     67      0110  0111
```

CF = 0 since there is no carry beyond d7
AF = 1 since there is a carry from d3 to d4
PF = 0 since there is an odd number of 1s in the result
ZF = 0 since the result is not zero
SF = 0 since d7 of the result is zero

Flag register and ADD instruction

In this section we examine the impact of the ADD instruction on the flag register, as an example of the use of the flag bits. The flag bits affected by the ADD instruction are CF (carry flag), PF (parity flag), AF (auxiliary carry flag), ZF (zero flag), SF (sign flag), and OF (overflow flag). The overflow flag will be covered in Chapter 6, since it relates only to signed number arithmetic. To understand how each of these flag bits is affected, look at Examples 1-10 and 1-11.

The same concepts apply for 16-bit addition, as shown in Examples 1-12 and 1-13. It is important to notice the differences between 8-bit and 16-bit operations in terms of their impact on the flag bits. The parity bit only counts the lower 8 bits of the result and is set accordingly. Also notice the CF bit. The carry flag is set if there is a carry beyond bit d15 instead of bit d7.

Notice the zero flag (ZF) status after the execution of the ADD instruction. Since the result of the entire 16-bit operation is zero (meaning the contents of BX), ZF is set to

Example 1-11

Show how the flag register is affected by

```
MOV   AL,9CH      ;AL=9CH
MOV   DH,64H      ;DH=64H
ADD   AL,DH       ;now AL=0
```

Solution:

```
      9C          1001  1100
+     64          0110  0100
      00          0000  0000
```

CF = 1 since there is a carry beyond d7
AF = 1 since there is a carry from d3 to d4
PF = 1 since there is an even number of 1s in the result
ZF = 1 since the result is zero
SF = 0 since d7 of the result is zero

Example 1-12

Show how the flag register is affected by

```
MOV   AX,34F5H    ;AX= 34F5H
ADD   AX,95EBH    ;now AX= CAE0H
```

Solution:

```
      34F5        0011  0100  1111  0101
+     95EB        1001  0101  1110  1011
      CAE0        1100  1010  1110  0000
```

CF = 0 since there is no carry beyond d15
AF = 1 since there is a carry from d3 to d4
PF = 0 since there is an odd number of 1s in the lower byte
ZF = 0 since the result is not zero
SF = 1 since d15 of the result is one

Example 1-13

Show how the flag register is affected by

```
MOV   BX,AAAAH    ;BX= AAAAH
ADD   BX,5556H    ;now BX= 0000H
```

Solution:

```
      AAAA        1010  1010  1010  1010
+     5556        0101  0101  0101  0110
      0000        0000  0000  0000  0000
```

CF = 1 since there is a carry beyond d15
AF = 1 since there is a carry from d3 to d4
PF = 1 since there is an even number of 1s in the lower byte
ZF = 1 since the result is zero
SF = 0 since d15 of the result is zero

Example 1-14

Show how the flag register is affected by

```
MOV   AX,94C2H    ;AX=94C2H
MOV   BX,323EH    ;BX=323EH
ADD   AX,BX       ;now AX=C700H
MOV   DX,AX       ;now DX=C700H
MOV   CX,DX       ;now CX=C700H
```

Solution:

```
      94C2        1001  0100  1100  0010
+     323E        0011  0010  0011  1110
      C700        1100  0111  0000  0000
```

After the ADD operation, the following are the flag bits:
CF = 0 since there is no carry beyond d15
AF = 1 since there is a carry from d3 to d4
PF = 1 since there is an even number of 1s in the lower byte
ZF = 0 since the result is not zero
SF = 1 since d15 of the result is 1

high. Do all instructions affect the flag bits? The answer is no; some instructions such as data transfers (MOV) affect no flags. As an exercise, run these examples on DEBUG to see the effect of various instructions on the flag register.

Running the instructions in Example 1-14 in DEBUG will verify that MOV instructions have no effect on the flag. How these flag bits are used in programming is discussed in future chapters in the context of many applications. In Appendix B we give additional information about the effect of various instructions on the flags.

Use of the zero flag for looping

One of the most widely used applications of the flag register is the use of the zero flag to implement program loops. The term *loop* refers to a set of instructions that is repeated a number of times. For example, to add 5 bytes of data, a counter can be used to keep track of how many times the loop needs to be repeated. Each time the addition is performed the counter is decremented and the zero flag is checked. When the counter becomes zero, the zero flag is set (ZF = 1) and the loop is stopped. The following shows the implementation of the looping concept in the program, which adds 5 bytes of data. Register CX is used to hold the counter and BX is the offset pointer (SI or DI could have been used instead). AL is initialized before the start of the loop. In each iteration, ZF is checked by the JNZ instruction. JNZ stands for "Jump Not Zero" meaning that if ZF = 0, jump to a new address. If ZF = 1, the jump is not performed and the instruction below the jump will be executed. Notice that the JNZ instruction must come immediately after the instruction that decrements CX since JNZ needs to check the effect of "DEC CX" on ZF. If any instruction were placed between them, that instruction might affect the zero flag.

```
          MOV   CX,05     ;CX holds the loop count
          MOV   BX,0200H  ;BX holds the offset data address
          MOV   AL,00     ;initialize AL
ADD_LP:   ADD   AL,[ BX]  ;add the next byte to AL
          INC   BX        ;increment the data pointer
          DEC   CX        ;decrement the loop counter
          JNZ   ADD_LP    ;jump to next iteration if counter not zero
```

Review Questions

1. The ADD instruction can affect which bits of the flag register?
2. The carry flag will be set to 1 in an 8-bit ADD if there is a carry out from bit ___.
3. CF will be set to 1 in a 16-bit ADD if there is a carry out from bit ____.

SECTION 1.7: x86 ADDRESSING MODES

The CPU can access operands (data) in various ways, called addressing modes. The number of addressing modes is determined when the microprocessor is designed and cannot be changed. The x86 provides a total of seven distinct addressing modes:

1. register
2. immediate
3. direct
4. register indirect
5. based relative
6. indexed relative
7. based indexed relative

Each addressing mode is explained below, and application examples are given in later chapters. ADD and MOV instructions are used below to explain addressing modes.

Register addressing mode

The register addressing mode involves the use of registers to hold the data to be manipulated. Memory is not accessed when this addressing mode is executed; therefore, it is relatively fast. Examples of register addressing mode follow:

```
MOV   BX,DX  ;copy the contents of DX into BX
MOV   ES,AX  ;copy the contents of AX into ES
```

```
ADD  AL,BH  ;add the contents of BH to contents of AL
```

It should be noted that the source and destination registers must match in size. In other words, coding "MOV CL,AX" will give an error, since the source is a 16-bit register and the destination is an 8-bit register.

Immediate addressing mode

In the immediate addressing mode, the source operand is a constant. In immediate addressing mode, as the name implies, when the instruction is assembled, the operand comes immediately after the opcode. For this reason, this addressing mode executes quickly. However, in programming it has limited use. Immediate addressing mode can be used to load information into any of the registers except the segment registers and flag registers. Examples:

```
MOV  AX,2550H      ;move 2550H into AX
MOV CX,625         ;load the decimal value 625 into CX
MOV BL,40H         ;load 40H into BL
```

To move information to the segment registers, the data must first be moved to a general-purpose register and then to the segment register. Example:

```
MOV  AX,2550H
MOV  DS,AX
MOV  DS,0123H ;illegal! cannot move data into segment reg.
```

In the first two addressing modes, the operands are either inside the microprocessor or tagged along with the instruction. In most programs, the data to be processed is often in some memory location outside the CPU. There are many ways of accessing the data in the data segment. The following describes those different methods.

Direct addressing mode

In the direct addressing mode the data is in some memory location(s) and the address of the data in memory comes immediately after the instruction. Note that in immediate addressing, the operand itself is provided with the instruction, whereas in direct addressing mode, the address of the operand is provided with the instruction. This address is the offset address and one can calculate the physical address by shifting left the DS register and adding it to the offset as follows:

```
MOV DL,[2400]  ;move contents of DS:2400H into DL
```

In this case the physical address is calculated by combining the contents of offset location 2400 with DS, the data segment register. Notice the bracket around the address. In the absence of this bracket executing the command will give an error since it is interpreted to move the value 2400 (16-bit data) into register DL, an 8-bit register. See Example 1-15.

Example 1-15

Find the physical address of the memory location and its contents after the execution of the following, assuming that DS = 1512H.

```
MOV   AL,99H
MOV   [3518],AL
```

Solution:
First AL is initialized to 99H, then in line two, the contents of AL are moved to logical address DS:3518, which is 1512:3518. Shifting DS left and adding it to the offset gives the physical address of 18638H (15120H + 3518H = 18638H). That means after the execution of the second instruction, the memory location with address 18638H will contain the value 99H.

Register indirect addressing mode

In the register indirect addressing mode, the address of the memory location where the operand resides is held by a register. The registers used for this purpose are SI, DI, and BX. If these three registers are used as pointers, that is, if they hold the offset of the memory location, they must be combined with DS in order to generate the 20-bit physical address. For example:

```
MOV AL,[ BX]  ;moves into AL the contents of the memory
              ;location pointed to by DS:BX.
```

Notice that BX is in brackets. In the absence of brackets, the code is interpreted as an instruction moving the contents of register BX to AL (which gives an error because source and destination do not match) instead of the contents of the memory location whose offset address is in BX. The physical address is calculated by shifting DS left one hex position and adding BX to it. The same rules apply when using register SI or DI.

```
MOV   CL,[ SI]        ;move contents of DS:SI into CL
MOV   [ DI] ,AH       ;move contents of AH into DS:DI
```

The examples above moved byte-sized data. Example 1-16 shows 16-bit data.

Example 1-16

Assume that DS = 1120, SI = 2498, and AX = 17FE. Show the contents of memory locations after the execution of "`MOV [ SI] ,AX`".

Solution:
The contents of AX are moved into memory locations with logical address DS:SI and DS:SI + 1; therefore, the physical address starts at DS (shifted left) + SI = 13698. According to the little endian convention, low address 13698H contains FE, the low byte, and high address 13699H will contain 17, the high byte.

Based relative addressing mode

In the based relative addressing mode, base registers BX and BP, as well as a displacement value, are used to calculate what is called the *effective address*. The default segments used for the calculation of the physical address (PA) are DS for BX and SS for BP. For example:

```
MOV CX,[ BX] +10     ;move DS:BX+10 and DS:BX+10+1 into CX
                     ;PA = DS (shifted left) + BX + 10
```

Alternative codings are "`MOV CX,[ BX+10]`" or "`MOV CX,10[ BX]`". Again the low address contents will go into CL and the high address contents into CH. In the case of the BP register,

```
MOV   AL,[ BP] +5     ;PA = SS (shifted left) + BP + 5
```

Again, alternative codings are "`MOV AL,[ BP+5]`" or "`MOV AL,5[ BP]`". A brief mention should be made of the terminology *effective address* used in Intel literature. In "`MOV AL,[ BP] +5`", BP+5 is called the effective address since the fifth byte from the beginning of the offset BP is moved to register AL. Similarly in "`MOV CX,[ BX] +10`", BX+10 is called the effective address.

Indexed relative addressing mode

The indexed relative addressing mode works the same as the based relative addressing mode, except that registers DI and SI hold the offset address. Examples:

```
MOV   DX,[ SI] +5     ;PA = DS (shifted left) + SI + 5
MOV   CL,[ DI] +20    ;PA = DS (shifted left) + DI + 20
```

Example 1-17 gives further examples of indexed relative addressing mode.

Example 1-17

Assume that DS = 4500, SS = 2000, BX = 2100, SI = 1486, DI = 8500, BP = 7814, and AX = 2512. All values are in hex. Show the exact physical memory location where AX is stored in each of the following. All values are in hex.
(a) `MOV[ BX] +20,AX` (b) `MOV[ SI] +10,AX`
(c) `MOV[ DI] +4,AX` (d) `MOV[ BP] +12,AX`

Solution:
In each case PA = segment register (shifted left) + offset register + displacement.
(a) DS:BX+20 location 47120 = (12) and 47121 = (25)
(b) DS:SI+10 location 46496 = (12) and 46497 = (25)
(c) DS:DI+4 location 4D504 = (12) and 4D505 = (25)
(d) SS:BP+12 location 27826 = (12) and 27827 = (25)

Based indexed addressing mode

By combining based and indexed addressing modes, a new addressing mode is derived called the *based indexed addressing mode.* In this mode, one base register and one index register are used. Examples:

```
MOV   CL,[ BX][ DI] +8   ;PA = DS (shifted left) + BX + DI + 8
MOV   CH,[ BX][ SI] +20  ;PA = DS (shifted left) + BX + SI + 20
MOV   AH,[ BP][ DI] +12  ;PA = SS (shifted left) + BP + DI + 12
MOV   AH,[ BP][ SI] +29  ;PA = SS (shifted left) + BP + SI + 29
```

The coding of the instructions above can vary; for example, the last example could have been written in either of the following two ways:

```
MOV   AH,[ BP+SI+29]
MOV   AH,[ SI+BP+29]   ;the register order does not matter
```

Note that "`MOV AX,[ SI][ DI] +displacement`" is illegal.

Table 1-5: Offset Registers for Various Segments

Segment register:	CS	DS	ES	SS
Offset register(s):	IP	SI, DI, BX	SI, DI, BX	SP, BP

In many of the examples above, the MOV instruction was used for the sake of clarity, even though one can use any instruction as long as that instruction supports the addressing mode. For example, the instruction "`ADD DL,[ BX]` " would add the contents of the memory location pointed at by DS:BX to the contents of register DL.

Segment overrides

Table 1-5 summarizes the offset registers that can be used with the four segment registers. The x86 CPU allows the program to override the default segment and use any segment register. To do that, specify the segment in the code. For example, in "`MOV AL,[ BX]` ", the physical address of the operand to be moved into AL is DS:BX, as was shown earlier since DS is the default segment for pointer BX. To override that default, specify the desired segment in the instruction as "`MOV AL,ES:[ BX]` ". Now the address of the operand being moved to AL is ES:BX instead of DS:BX. Extensive use of all these

addressing modes is shown in future chapters in the context of program examples.

Table 1-6 shows more examples of segment overrides shown next to the default address in the absence of the override. Table 1-7 summarizes addressing modes of the 8088/86.

Table 1-6: Sample Segment Overrides

Instruction	Segment Used	Default Segment
MOV AX, CS:[BP]	CS:BP	SS:BP
MOV DX,SS:[SI]	SS:SI	DS:SI
MOV AX,DS:[BP]	DS:BP	SS:BP
MOV CX,ES:[BX]+12	ES:BX+12	DS:BX+12
MOV SS:[BX][DI]+32,AX	SS:BX+DI+32	DS:BX+DI+32

Table 1-7: Summary of the x86 Addressing Modes

Addressing Mode	Operand	Default Segment
Register	reg	none
Immediate	data	none
Direct	[offset]	DS
Register indirect	[BX]	DS
	[SI]	DS
	[DI]	DS
Based relative	[BX]+disp	DS
	[BP]+disp	SS
Indexed relative	[DI]+disp	DS
	[SI]+disp	SS
Based indexed relative	[BX][SI]+disp	DS
	[BX][DI]+disp	DS
	[BP][SI]+disp	SS
	[BP][DI]+disp	SS

Review Questions

1. Can the x86 programmer make up new addressing modes?
2. Is the IP (instruction pointer) register also available in low-byte and high-byte formats?
3. Is the CS (code segment) register also available in low-byte and high-byte formats?
4. Which segment is used for the direct addressing mode?
5. Which registers can be used for the register indirect addressing mode?

PROBLEMS

SECTION 1.1: BRIEF HISTORY OF THE x86 FAMILY

1. Which microprocessor, the 8088 or the 8086, was released first?
2. If the 80286 and 80386SX both have 16-bit external data buses, what is the difference between them?
3. What does "16-bit" or "32-bit" microprocessor mean? Does it refer to the internal or external data path?
4. Do programs written for the 8088/86 run on 80286-, 80386-, and 80486-based CPUs?
5. What does the term *upward compatibility* mean?

6. Name a major difference between the 8088 and the 8086.
7. Which has the larger queue, the 8088 or the 8086?

SECTION 1.2: INSIDE THE 8088/86

8. State another way to increase the processing power of the CPU other than increasing the frequency.
9. What do "BIU" and "EU" stand for, and what are their functions?
10. Name the general-purpose registers of the 8088/86.
    (a) 8-bit (b) 16-bit
11. Which of the following registers cannot be split into high and low bytes?
    (a) CS (b) AX (c) DS
    (d) SS (e) BX (f) DX
    (g) CX (h) SI (i) DI

SECTION 1.3: INTRODUCTION TO ASSEMBLY PROGRAMMING

12. Which of the following instructions cannot be coded in 8088/86 Assembly language? Give the reason why not, if any. To verify your answer, code each in DEBUG. Assume that all numbers are in hex.
    (a) MOV AX,27 (b) MOV AL,97F (c) MOV DS,9BF2
    (d) MOV CX,397 (e) MOV SI,9516 (f) MOV CS,3490
    (g) MOV DS,BX (h) MOV BX,CS (i) MOV CH,AX
    (j) MOV AX,23FB9 (k) MOV CS,BH (l) MOV AX,DL

SECTION 1.4: INTRODUCTION TO PROGRAM SEGMENTS

13. Name the segment registers and their functions in the 8088/86.
14. If CS = 3499H and IP = 2500H, find:
    (a) The logical address (b) The physical address
    (c) The lower and upper ranges of the code segment
15. Repeat Problem 14 with CS = 1296H and IP = 100H.
16. If DS = 3499H and the offset = 3FB9H, find:
    (a) The physical address (b) The logical address of the data being fetched
    (c) The lower and upper range addresses of the data segment
17. Repeat Problem 16 using DS = 1298H and the offset = 7CC8H.
18. Assume that the physical address for a location is 0046CH. Suggest a possible logical address.
19. If an instruction that needs to be fetched is in physical memory location 389F2 and CS = 2700, does the code segment range include it or not? If not, what value should be assigned to CS if the IP must equal 1282?
20. Using DEBUG, assemble and unassemble the following program and provide the logical address, physical address, and the content of each address location. The CS value is decided by Windows, but use IP = 170H.

```
MOV AL,76H
MOV BH,8FH
ADD BH,AL
ADD BH,7BH
MOV BL,BH
ADD BL,AL
```

21. Repeat Problem 20 for the following program from page 36.

```
MOV AL,0       ;clear AL
ADD AL,[0200]   ;add the contents of DS:200 to AL
ADD AL,[0201]   ;add the contents of DS:201 to AL
ADD AL,[0202]   ;add the contents of DS:202 to AL
ADD AL,[0203]   ;add the contents of DS:203 to AL
ADD AL,[0204]   ;add the contents of DS:204 to AL
```

SECTION 1.5: THE STACK

22. The stack is:
    (a) A section of ROM
    (b) A section of RAM used for temporary storage
    (c) A 16-bit register inside the CPU
    (d) Some memory inside the CPU
23. In problem 22, choose the correct answer for the stack pointer.
24. When data is pushed onto the stack, the stack pointer is _____, but when data is popped off the stack, the stack pointer is _____.
25. Choose the correct statement:
    (a) The stack segment and code segment start at the same point of read/write memory and grow upward.
    (b) The stack segment and code segment start at opposite points of read/write memory and grow toward each other.
    (c) There will be no problem if the stack and code segments meet each other.
26. What is the main disadvantage of the stack as temporary storage compared to having a large number of registers inside the CPU?
27. If SS = 2000 and SP = 4578, find:
    (a) The physical address
    (b) The logical address
    (c) The lower range of the stack segment
    (d) The upper range of the stack segment
28. If SP = 24FC, what is the offset address of the first location of the stack that is available to push data into?
29. Assume that SP = FF2EH, AX = 3291H, BX = F43CH, and CX = 09. Find the contents of the stack and SP after the execution of each of the following instructions.

```
PUSH   AX
PUSH   BX
PUSH   CX
```

30. Show the sequence of instructions needed to restore all the registers to their original values in Problem 29. Show the content of SP at each point.
31. The following registers are used as offsets. Assuming that the default segment is used to get the logical address, give the segment register associated with each offset.
    (a) BP (b) DI (c) IP
    (d) SI (e) SP (f) BX
32. Show the override segment register and the default segment register used (if there were no override) in each of the following cases.
    (a) MOV SS:[BX],AX (b) MOV SS:[DI],BX
    (c) MOV DX,DS:[BP+6]

SECTION 1.6: FLAG REGISTER

33. Find the status of the CF, PF, AF, ZF, and SF for the following operations.
    (a)MOV BL,9FH (b) MOV AL,23H (c) MOV DX,10FFH
    ADD BL,61H ADD AL,97H ADD DX,1

SECTION 1.7: x86 ADDRESSING MODES

34. Assume that the registers have the following values (all in hex) and that CS = 1000, DS = 2000, SS = 3000, SI = 4000, DI = 5000, BX = 6080, BP = 7000, AX = 25FF, CX = 8791, and DX = 1299. Calculate the physical address of the memory where the operand is stored and the contents of the memory locations in each of the following addressing examples.
    (a) MOV [SI],AL (b) MOV [SI+BX+8],AH
    (c) MOV [BX],AX (d) MOV [DI+6],BX
    (e) MOV [DI][BX]+28,CX (f) MOV [BP][SI]+10,DX

(g) MOV [3600],AX
(h) MOV [BX]+30,DX
(i) MOV [BP]+200,AX
(j) MOV [BP+SI+100],BX
(k) MOV [SI]+50,AH
(l) MOV [DI+BP+100],AX.

35. Give the addressing mode for each of the following:
(a) MOV AX,DS
(b) MOV BX,5678H
(c) MOV CX,[3000]
(d) MOV AL,CH
(e) MOV [DI],BX
(f) MOV AL,[BX]
(g) MOV DX,[BP+DI+4]
(h) MOV CX,DS
(i) MOV [BP+6],AL
(j) MOV AH,[BX+SI+50]
(k) MOV BL,[SI]+10
(l) MOV [BP][SI]+12,AX

36. Show the contents of the memory locations after the execution of each instruction.
(a) MOV BX,129FH
MOV [1450],BX
DS:1450 ....
DS:1451 ....
(b) MOV DX,8C63H
MOV [2348],DX
DS:2348 ....
DS:2349 ....

ANSWERS TO REVIEW QUESTIONS

SECTION 1.1: BRIEF HISTORY OF THE x86 FAMILY

1. (1) increased memory capacity from 64K to 1 megabyte;
(2) the 8086 is a 16-bit microprocessor instead of an 8-bit microprocessor;
(3) the 8086 was a pipelined processor
2. The 8088 has an 8-bit external data bus whereas the 8086 has a 16-bit data bus.
3. (a) 20-bit, 1 megabyte (b) 24-bit, 16 megabytes (c) 32-bit, 4 gigabytes
4. 16, 32
5. The 80386 has 32-bit address and data buses, whereas the 80386SX has a 24-bit address bus and a 16-bit external data bus.
6. Virtual memory, protected mode
7. Math coprocessor on the CPU chip, cache memory and controller
8. A 64-bit bus, faster floating point, and separate cache memory for code and data.
9. 7.5 million
10. Pentium II
11. MMX
12. 8-bit, 16-bit, and 32-bit
13. 8-bit, 16-bit, 32-bit, and 64-bit
14. True

SECTION 1.2: INSIDE THE 8088/86

1. The execution unit executes instructions; the bus interface unit fetches instructions.
2. Pipelining divides the microprocessor into two sections: the execution unit and the bus interface unit; this allows the CPU to perform these two functions simultaneously; that is, the BIU can fetch instructions while the EU executes the instructions previously fetched.
3. 8, 16
4. AX, BX, CX, DX, SP, BP, SI, DI, CS, DS, SS, ES, IP, FR

SECTION 1.3: INTRODUCTION TO ASSEMBLY PROGRAMMING

1. MOV BX,1234H
2. MOV AX,16H
ADD AX,ABH
3. The segment registers CS, DS, ES, and SS
4. FFFFH = 6553510, FFH = 25510

SECTION 1.4: INTRODUCTION TO PROGRAM SEGMENTS

1. 64K
2. A segment contains 64K bytes; yes because 346E0H is evenly divisible by 16.
3. The physical address is the 20-bit address that is put on the address bus to locate a byte; the logical address is the address in the form xxxx:yyyy, where xxxx is the segment address and yyyy is the offset into the segment.
4. 20, 16
5. IP
6. 2400 would contain 34 and 2401 would contain 12.

SECTION 1.5: THE STACK

1. SS is the segment register; SP and BP are used as pointers into the stack.
2. Decremented
3. Incremented
4. 143F:0000, 1000:43F0, 1410:02F0

SECTION 1.6: FLAG REGISTER

1. CF, PF, AF, ZF, SF, and OF
2. 7
3. 15

SECTION 1.7: x86 ADDRESSING MODES

1. No
2. No
3. No
4. DS and ES
5. BX, SI, and DI