1 Introducing the 8086 Microprocessor

LEARNING OBJECTIVES

At the end of this chapter, you will be able to:

- Identify the major functional units of a microprocessor and microcomputer and list their functions.
- List the features of the 8086 microprocessor.
- Draw the logical block diagram of the Intel 8086 microprocessor.
- List the functions and parts of the Bus Interface Unit and the Execution Unit of the 8086 microprocessor.
- List the various types of registers with their meaning and use.
- List the various conditional and control flags with their meaning and use.
- List the segment registers with their meaning.
- Explain the generation of physical address using segment base and the offset.
- Draw the pin diagram of the 8086 microprocessor and list the functions of each pin.
- Compare the minimum and maximum mode.
- Explain the memory system supported by the 8086 microprocessor.

1.1 Introduction

A microprocessor is an Integrated Chip produced using the VLSI (Very Large Scale Integration) technology which acts as the heart of the modern microcomputer systems and performs all the computations. The microprocessors have evolved like revolution since the development of first 4-bit microprocessor 4004 by Intel in 1971. The microprocessors have improved in all the fronts including speed, processing capabilities, memory, powerful instructions and many more in last 40 years. Today, microprocessors are used to build from small dedicated systems for industrial applications to general-purpose computer systems used for running large and complex software including business applications to scientific applications.

Intel's 8-bit microprocessors (8008, 8080, 8085) proved the commercial success of the microprocessor technology. Intel's 16-bit microprocessor 8086 is a result of such revolution and development. The 8086 divides its internal architecture into two hardware divisions and uses the technique like prefetch queue for improved performance. The support for segmentation allows program components such as code, data and stack to be in separate memory areas called *segments* for better management. The 8086 operates in two different hardware modes for different system requirements:

maximum mode and minimum mode. The 8086 supports 1 MB memory using the concept of memory banks: odd and even banks. All the above features of the 8086 are explored in this chapter in detail.

1.2 Brief History of Intel Microprocessors

The journey of Intel microprocessors started with the announcement of its first 4-bit microprocessor in 1971. The numbers of bits represent the size of the operands the ALU of the processor can process as well as the width of its data bus. The 4004 microprocessor was designed to be used in calculators. It was supporting only 640 bytes of addressable memory. Although the 4004 was having limited capabilities, it was the first ever milestone in the history of Intel microprocessors. The success of the 4004 microprocessor motivated the Intel team to come out with a more capable microprocessor 8008 in 1972. The 8008 was the first 8-bit microprocessor with support of the 16 KB addressable memory. It was originally designed to be used in microcomputers. The Intel then came out with the improved versions of the 8008; the 8080 microprocessor in 1974 and the 8085 microprocessor in 1976. The 8080 and 8085 both were 8-bit microprocessors having 8-bit data bus and 16-bit address bus addressing total 64 KB memory. The major improvement from the 8008 to 8080 was the reduction in number of support chips to form the functional CPU. The 8008 required 20 support chips whereas 8080 required only 6 support chips. This was the result of the high-level integration of the components in a chip. The 8085 used a single +5 V power supply against the +12 V power supply used in previous versions. Each of the microprocessor in the 8-bit series also improved in clock speed than its predecessor. The 8085 was commercially a very successful 8-bit microprocessor from Intel and attracted the industry's attention for the use in industrial applications. Today, even the 8085 is in use in the dedicated industrial applications.

Intel came out with 16-bit microprocessor 8086 in 1978 with many more features than its 8-bit series. The 20 address lines in 8086 made it capable of addressing 1 MB memory, much higher than 64 KB in the 8085. The 8086 memory is divided into two memory banks – odd and even banks – and allows 16-bit data to be read at a time. It also supports segmentation with each segment having maximum size of 64 KB. The 8086 was mainly used in portable computing. It was also used in IBM PS/2 Model 25 and Model 30. Intel produced the variant of the 8086 called 8088 in 1979. The internal architecture of the 8088 is fully 16-bit as like the 8086 with 20 address lines. Only the external data bus was 8-bit as against 16-bit in the 8086. Due to this variation in data bus size, the 8086 is referred as internally and externally as 16-bit microprocessor whereas the 8088 is referred as internally 16-bit but externally 8-bit microprocessor. The 8086 and 8088 are both software compatible. The 8088 was used as the processor in the IBM PC machines (and their clones) which made revolution in computing history.

Intel subsequently came out with the more advanced microprocessors 80826, 80386, 80486 and Pentium (80586) series, each of them much capable than their predecessors. The whole series, right from 8086 to Pentium, is popularly known as the *80X86 series* of processors. The IBM PC and the compatible machines were built around these processors. Currently, higher and improved versions of the Pentium series processors are dominating the desktop computer market. We will study the architectures of the 80286, 80386 and 80486 in Chapters 8, 9 and 10. The overview of the Pentium series of processors is covered in Chapter 11.

Check Points

We now know that

- the 4-bit microprocessor 4004 was Intel's first microprocessor.
- the 8008, 8080 and 8085 are Intel's 8-bit microprocessors.
- the 8086 and 8088 are Intel's 16-bit microprocessors.
- the IBM PC and compatible machines use 8088 microprocessor.
- the 80X86 series of microprocessors include 8086, 80286, 80386, 80486 and Pentium series of microprocessors.
- the modern desktop machines use 80X86 processors.

1.3 Overview of Microprocessor and Microcomputer

A microcomputer system is designed around a microprocessor with other peripheral chips in order to connect the I/O (Input/Output) devices like keyboard, display, disk drives, printer, etc. and the memory. Let us take an overview of the microprocessor and microcomputer in general to get an idea of how the microprocessor works and the microcomputer systems are designed using microprocessor as central device.

Microprocessor

We know that microprocessor is an integrated chip responsible for performing all the computations including arithmetic and logical operations. Figure 1 shows the view of a general microprocessor.

As can be seen from the Fig. 1, the microprocessor contains three internal parts: register array, ALU (Arithmetic and Logical Unit) and control and timing. The ALU is a digital circuit responsible for performing all the arithmetic and logical operations. The size of the ALU determines the size of the operands to be processed. For example, 16-bit ALU performs the operation on 16-bit operands. The ALU size normally is 8-bit, 16-bit, 32-bit or 64-bit. The ALU needs the operands to be in internal registers. The register array provides the internal memory for the ALU to store operands and contains a set of registers. The registers in the register array are classified according to their purpose. The common types are data, address and flag registers. The microprocessor is interfaced with external devices to make

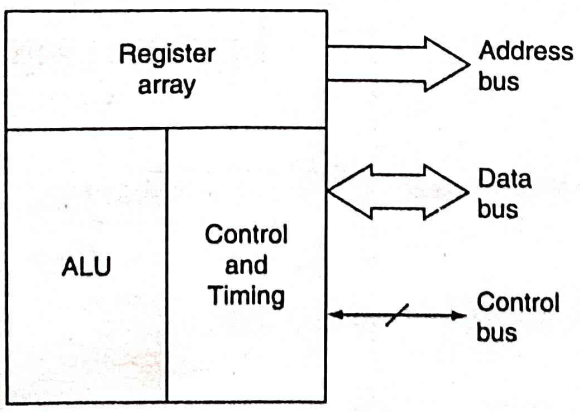


Figure 1 View of a general microprocessor.

a microcomputer system. The control and timing section controls all the operations performed by the microprocessor and provides the synchronization in order to perform the operations with the help of the external devices. The major functions of the microprocessor are listed as follows.

1. Execute a program instruction by instruction. Executing an instruction is called *instruction cycle* which consists of three phases: opecode fetch, decode and execute. The opcode fetch reads the instruction from the memory, the decode phase determines the meaning of the instruction and finally the execute phase performs necessary actions to complete the execution of the instruction. This is also called *fetch-decode-execute* cycle. The microprocessor repeatedly performs this cycle unless it is stopped.
2. Perform the data transfer using the data bus between itself and memory or I/O devices. This provides the movement of data within the system.
3. Provide the overall control in the system and synchronization in order to perform the operations successfully.
4. Provide the services to the external devices by accepting the interrupt requests through the interrupt lines and executing their service routines.

The complexity of a microprocessor depends upon the complexity of its internal parts. All the parts of the advanced processors are much complex. For example, the 80286 is much more complex than 8086. The same is true for Pentium as compared to 80286.

The microprocessor is interfaced with the other external devices using its data, address and control buses. These three buses together are called *system bus*. The data bus is bidirectional and is used to transfer data between microprocessor and memory or I/O devices. The size of the data bus determines the numbers of bits moved at a time. The address bus is unidirectional and provides the addresses for memory locations and I/O devices. The numbers of address lines determine the size of the memory a microprocessor can address. The control bus consists of individual or set of control signals to synchronize with the external devices. The size of the buses depends again on the complexity of the microprocessor.

Microcomputer

A microcomputer is a digital system consisting of microprocessor, memory and I/O devices connected using system bus. The system bus includes data, address and control buses. Figure 2 shows the block diagram of a microcomputer system.

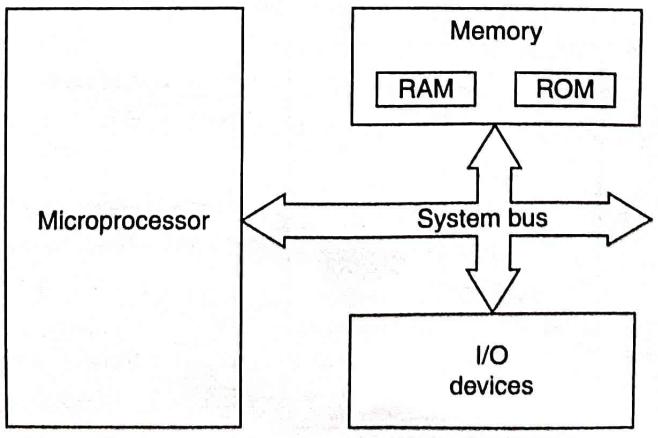


Figure 2 Block diagram of a microcomputer system.

The microprocessor acts as a central device and communicates with the memory and I/O devices using the system bus. The input devices are used to get data from the external world and the output devices are used to send the data to the outside world. Storage devices such as hard disks act as both input and output devices and form the secondary storage. The memory forms the primary storage which includes both RAM (Random Access Memory) and ROM (Read Only Memory). The ROM stores the permanent programs whereas RAM provides the space for user programs.

Check Points

We now know that

- a microprocessor is an integrated chip responsible for performing all the computations.
- the microprocessor is internally divided into register array, ALU and control and timing parts.
- the registers provide internal memory, ALU performs the arithmetic and logic operations and control and timing provide the control and synchronization.
- the microprocessor works on the philosophy of fetch-decode-execute. The other functions of microprocessor are data transfer, control and synchronization and interrupt services to the external devices.
- the microprocessor contains data, address and control bus together called system bus to interface it with other devices.
- a microcomputer is a digital system consisting of microprocessor, memory and I/O devices connected using system bus.

1.4 Features of the 8086 Microprocessor

The 8086 is Intel's first 16-bit microprocessor in 80X86 series with a lot of improvement compared to its 8-bit series. The following are the major features of the 8086 microprocessor:

- 40 pin DIP (Dual Inline Package).
- 16-bit ALU.
- 16-bit data bus.
- 20-bit address bus, total addressable memory $2^{20} = 1$ MB.
- Two internal hardware units :
  - BIU (Bus Interface Unit).
  - EU (Execution Unit).
- 6 byte prefetch queue.
- Four 16-bit general-purpose registers (AX, BX, CX and DX) also accessible as eight 8-bit registers (AH, AL, BH, BL, CH, CL, DH and DL), two 16-bit index registers (SI and DI), two 16-bit stack pointers (SP and BP).
- Four segment registers (CS, DS, ES and SS) and an Instruction Pointer (IP) register, each 16-bit.
- 16-bit flag register with six conditional flags (OF, SF, ZF, PF, AF and CF) and three control flags (DF, IF and TF).
- Two hardware modes: Maximum and minimum mode.

- Memory divided into odd and even banks and accessible simultaneously to read 16-bit word in one cycle (if word starts from even address).
- Instruction set supporting
  - Variety of flexible addressing modes.
  - Multiplication and division instructions.
  - Special set of string instructions and repeat prefixes.
  - Special iteration control instructions.
  - Signed and unsigned processing.
  - Binary, BCD and ASCII processing.

The 8088 supports same features except the external data bus which is 8-bit (reads word always in two cycles) and prefetch queue which is of 4 bytes. The 8086 and 8088 are software compatible and program written for 8086 also runs on 8088. Throughout the discussion of 8086 architecture and programming, unless it is mentioned, the same is applied for 8088 also. The 8086/8088 processor is upward compatible to all the higher microprocessors (80286 to Pentium series) in 80X86 series and hence an 8086 program runs on any of them. The processor is a common term used for microprocessors and hence we will use terms processor and microprocessor interchangeably.

1.5 The 8086 Architecture

The internal architecture of the 8086 microprocessor consists of two separate processing units: Bus Interface Unit (BIU) and Execution Unit (EU). The logical block diagram showing the architecture of the 8086 microprocessor is given in Fig. 3. Both of the units have their own dedicated functions and operate in parallel. The 8086 makes the fetching and execution of the instructions independent of each other using BIU and EU to improve the overall performance. Let us understand the functions and components of BIU and EU.

Bus Interface Unit (BIU)

The BIU interfaces the 8086 microprocessor with the external devices and performs all the external operations using its 16-bit data bus, 20-bit address bus and various control signals. The main functions of the BIU are as follows.

1. Fetching the instructions from memory.
2. Read and write operands with memory and I/O devices.
3. Address generation.
4. Queuing of the prefetched instruction bytes.

The 8086 BIU computes the address, sends it on the address bus to fetch the instructions or to perform data transfer from and to memory or I/O devices. The major functional components of the BIU are discussed as follows.

Prefetch Queue

The 8086 uses the concepts of pipelining to provide the parallelism between the BIU and EU. While the EU is busy in executing an instruction, the BIU can fetch the next instruction. The question is where does the BIU store the prefetched instruction until it is read by the EU for decoding and

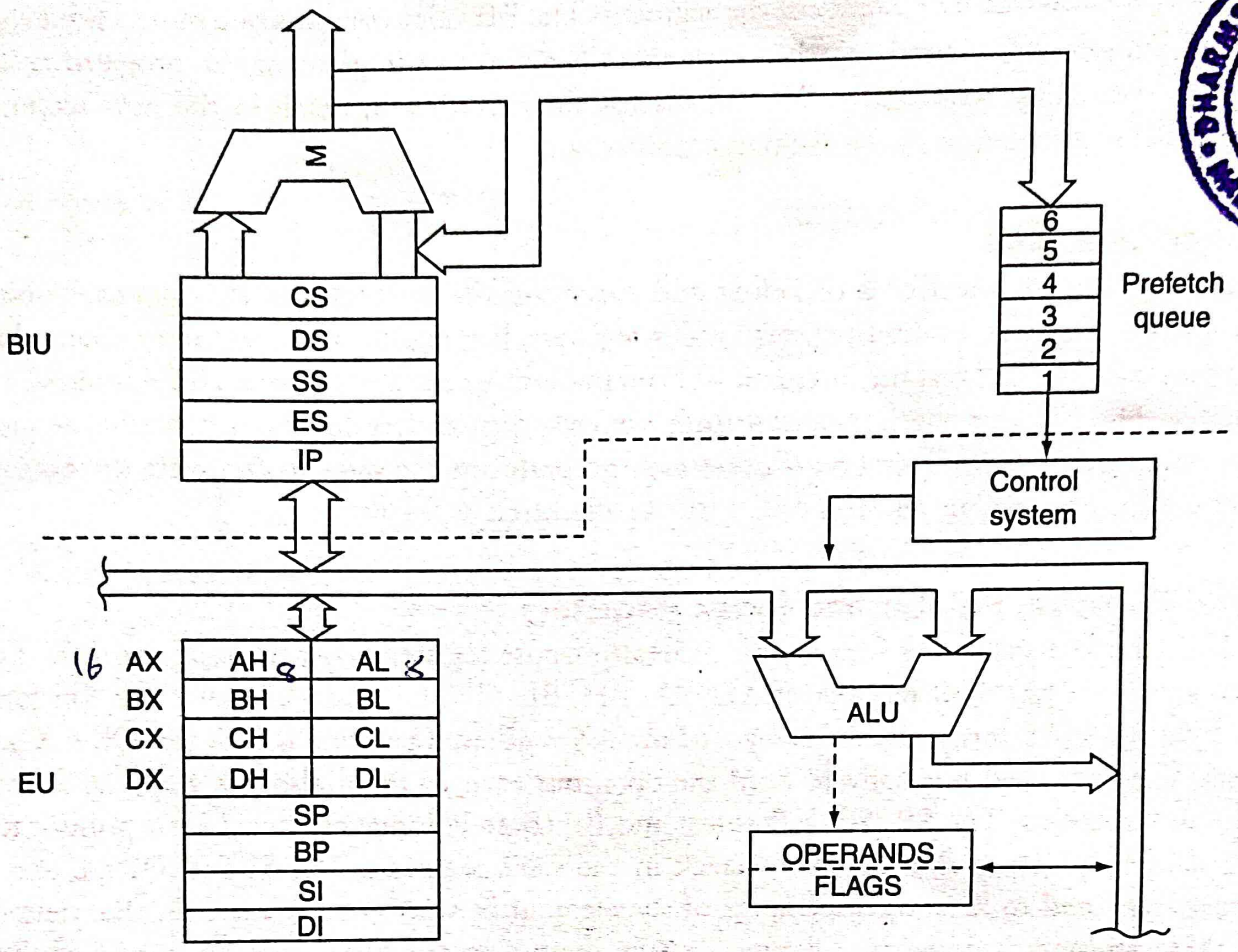


Figure 3 Architecture of Intel 8086 microprocessor.

execution? The 8086 BIU contains 6 bytes (4 bytes in 8088) in the first-in-first-out queue known as *prefetch or instruction queue* to store the instruction bytes fetched ahead of time. This helps in improving the performance as the EU need not wait for the next instruction to be fetched when it completes the execution of the previous instruction. The next instruction is already available in the queue. If the queue is not full and the BIU is free, then it fetches the instruction bytes from the memory and puts them into the queue. At the same time the EU is executing the instruction read previously from the queue. Thus, the instruction fetching and execution are made independent and carried out in parallel. This improves the performance of the 8086 (and 8088) significantly. The BIU fetches the instructions from the memory in physical order which creates a problem when the next instruction in the queue is branch control instruction, like JMP or CALL, as the further execution may not continue in sequence. The BIU flushes the queue in such a situation and fills the queue from the new address provided by the branch instruction to resolve the issue.

Segment Registers and Instruction Pointer

The 8086 can address total of 1 MB (2^{20} = 1 MB) memory using its 20-bit address bus. The 1 MB memory is divided into 64 KB chunks known as *segments*. At a time only one of the 64 KB chunks is accessible through the one of the four 16-bit segment registers which includes CS (Code Segment), DS (Data Segment), SS (Stack Segment) and ES (Extra Segment) registers. A segment register stores the

starting address known as *base address* of the segment. The BIU also contains one more 16-bit register – Instruction Pointer (IP) – used together with the CS register to implement the program counter to provide the instruction sequencing. We will discuss them further in detail in the next section while learning about segmentation in the 8086 microprocessor.

Execution Unit (EU)

The main function of the EU is decoding and executing the instructions. It contains 16-bit ALU, general-purpose registers, pointer registers, index registers, flag register and temporary operand register for that purpose. The EU gets the instructions from prefetch queue and operands from internal registers or memory. The EU gets the instruction from prefetch queue, decodes the instruction, requests the BIU for data read or write operation if necessary and performs the steps to complete the execution of the instruction. The various registers of the EU are discussed as follows.

General-Purpose, Pointer and Index Registers

The EU of the 8086 contains four 16-bit general-purpose registers: AX, BX, CX and DX. They are also accessible as eight 8-bit registers as AH, AL, BH, BL, CH, CL, DH and DL. The AH forms the higher byte while AL forms the lower byte of the AX register. The same is true for BX, CX and DX registers. They are used normally to hold the data, but each of them also has a special function in specified instructions. The SP (Stack Pointer) and BP (Base Pointer) are two 16-bit pointer registers mainly used to point to the top of the stack in the stack segment. The SI and DI are two 16-bit index registers used to keep track of index of the element in arrays and strings. Further detail about all of these registers is given in Chapter 2 while discussing the programming model of the 8086 microprocessor.

Flag Register

The 16-bit flag register of the 8086 microprocessor contains nine flags including six conditional flags and three control flags. Figure 4 shows the flag register of the 8086 microprocessor. Observe that the lower byte of the flag register contains Intel's 8-bit processor, 8085, compatible flags.

The six conditional flags include Carry Flag (CF), Parity Flag (PF), Auxiliary Carry Flag (AF), Zero Flag (ZF), Sign Flag (SF) and Overflow Flag (OF). The conditional flags are set or reset based on the result of execution of the arithmetic and logical instructions. This means that these six flags provide the status of the program. The meanings of the conditional flags are given in Table 1.

Let us take an example of adding two 16-bit numbers 8256h and F13Dh as follows.

```
    1000 0010 0101 0110 (8256h)
+   1111 0001 0011 1101 (F13Dh)
-------------------------------
  10111 0011 1001 0011
```

From the result, it is clear that CF = 1, PF = 1, AF = 1, ZF = 0, SF = 0 and OF = 1. For 8-bit addition, the flags are set/reset based on 8-bit result plus the overflow.

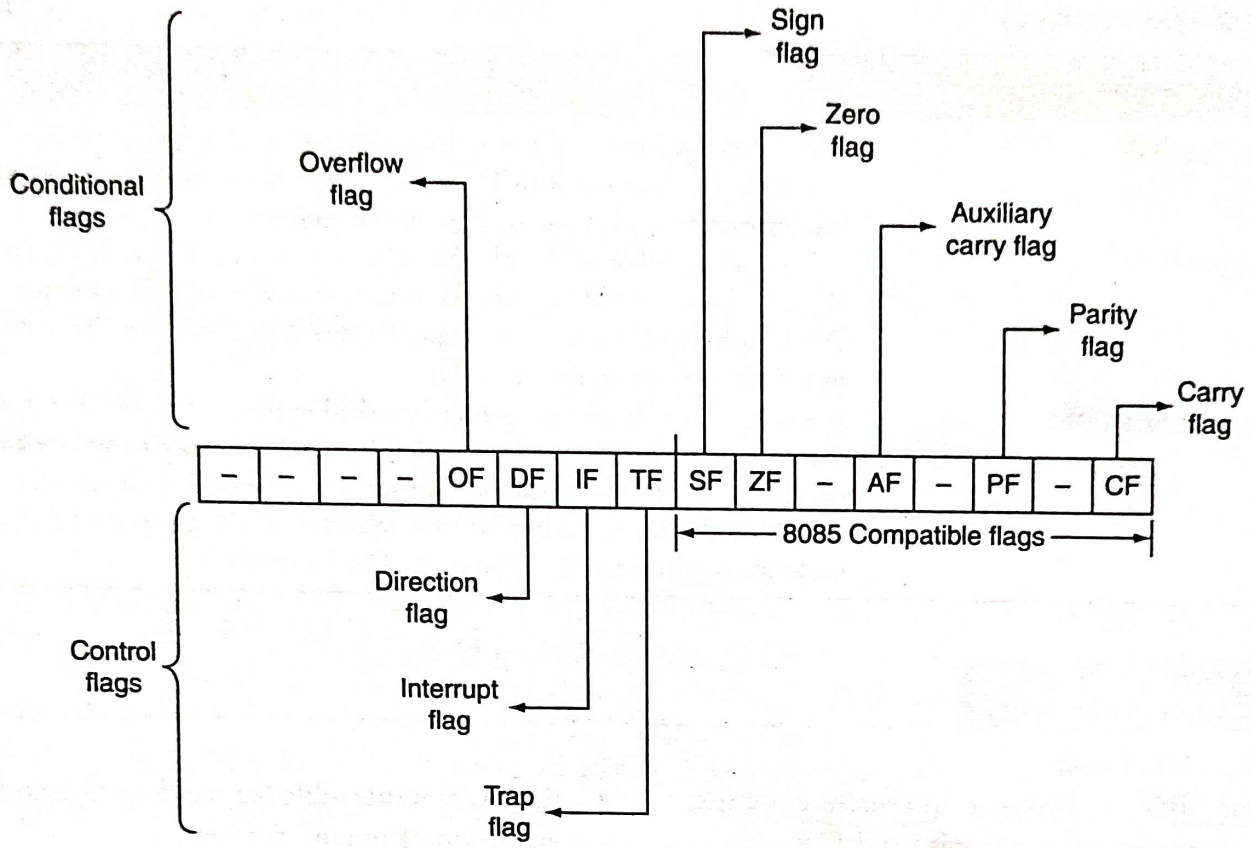


Figure 4 The 8086 flag register.

Table 1 Conditional flags

Conditional Flag	*Meaning*
Carry Flag (CF)	The CF stores the overflow of unsigned addition. If overflow is 1 then CF = 1, otherwise CF = 0. It works as borrow flag in subtraction.
Parity Flag (PF)	The PF stores the parity of the result. If parity is even then PF = 1, otherwise PF = 0 (odd parity).
Auxiliary Carry Flag (AF)	AF stores the overflow from the lower nibble of the addition. If overflow is 1 then AF = 1, otherwise AF = 0.
The Zero Flag (ZF)	The ZF stores the status of result whether it is zero or not. If the result is 0 then ZF = 1, otherwise ZF = 0.
Sign Flag (SF)	The SF stores the sign (MSB) of the result. If MSB of the result is 1 then SF = 1, otherwise SF = 0.
Overflow Flag (OF)	The OF stores the overflow of signed operation. If the result of the signed operation is out of range then OF = 1, otherwise OF = 0. For example, if the result of adding two 8-bit signed numbers is less than −128 or greater than +127 then the OF = 1.

The three control flags include the Trap Flag (TF), the Interrupt Flag (IF) and the Direction Flag (DF). The control flags are used to control the behaviour of the processor. The control flags and their description are given in Table 2.

Table 2 Control flags

Control Flag	*Meaning*
Trap Flag (TF)	The TF is used for single stepping and used by debugging tools. If the instruction is executed with TF = 1, the processor stops after execution of the instruction and allows user to see the registers and memory.
Interrupt Flag (IF)	It is used to enable or disable the maskable interrupts on INTR pin. If IF = 1, the processor accepts the interrupts on maskable interrupt pin INTR, otherwise it does not accept. The 8086 provides the STI and CLI instructions to set and reset the IF.
Direction Flag (DF)	It is used to decide the direction of processing the strings. If DF = 1 then the strings are processed from end-to-start, that is, addresses are auto-decremented. If DF = 0 then the strings are processed from start-to-end, that is, addresses are auto-incremented. The 8086 provides STD and CLD instructions to set and reset the DF, respectively.

Check Points

We now know that

- the 8086 architecture internally contains two processing units: BIU and EU.
- the BIU is responsible to interface the processor with external devices and performs the external operations.
- the main functions of BIU are instruction fetching, data read/write, address generation and instruction queuing.
- the BIU contains 6 byte instruction queue to store the instruction bytes fetched ahead of time.
- the BIU contains four segment registers and an instruction pointer.
- the EU is responsible for the decoding and execution of the instructions.
- the EU contains the ALU, four general-purpose registers, two pointer registers, two index registers and a flag register.
- the flag register contains six conditional flags and three control flags.
- the BIU and EU make the instruction fetching and execution independent and work in parallel.

1.6 Segmentation in the 8086 Microprocessor

The 20 address lines in the 8086 microprocessor make it capable of addressing 1 MB external memory with the address range 00000h to FFFFFh. The address range of the processor defines the memory map for the processor. The memory map of the 8086 (8088) microprocessor is shown in Fig. 5.

As can be seen from Fig. 5, the memory is organized in byte (8-bit data) sequence. The word (16-bit data) is stored in two consecutive locations with lower byte at lower address and the higher byte at higher address. If the word is aligned at even boundary, that is, starts from even address 00000h, 00002h, 00004h, etc., then it is accessible in one cycle. If the word is aligned at odd boundary, that is,

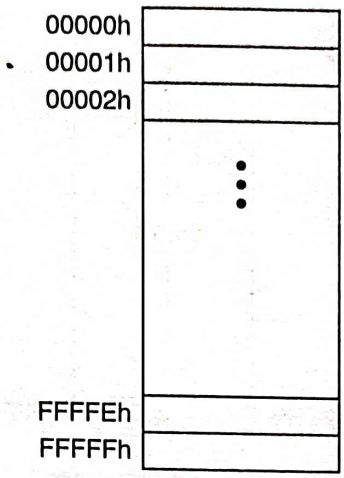


Figure 5 Memory map of the 8086 microprocessor.

starts from odd address 00001h, 00003h, 00005h, etc., then it is accessible in two cycles. We will see the reason for that while discussing about the 8086 memory banks. The double word (32-bit) data is stored in four consecutive bytes with lowest byte first and highest byte last.

Although there is 1 MB memory, only 64 KB memory is accessible at a time. The reason is that the 8086 memory is divided into chunks of 64 KB continuous locations known as *segments*. To access the memory covered in a segment, the segment registers store the starting address called *segment or base address*. The BIU of the 8086 microprocessor contains four 16-bit segment registers as listed below.

1. Code Segment (CS) register.
2. Data Segment (DS) register.
3. Stack Segment (SS) register.
4. Extra Segment (ES) register.

The 8086 allows only one of the four segments to remain active at a time and uses the contents of the segment registers as pointers to point to the starting of the respective segment. Each segment register is of 16-bit which makes the maximum size of a segment 64 KB (2^{16} = 64 KB). The segment register stores the starting address of the segment, but the question is how does a 16-bit register store 20-bit address? The 8086 divides the 20-bit physical address into two 16-bit parts called *segment* and *offset*, represented in the format **seg:offset**. The segment part is fixed for whole segment and stored in one of the segment registers whereas the offset part varies from 0000h to FFFFh. For example, the physical address 30525h is represented as 3052:0005h where 3052h is the segment address and 0005h is the offset address. The segment address is also known as the *segment base*. The offset address is the *distance or displacement* from the segment base. Figure 6 shows an 8086 segment starting from the 30520h. The offset address acts as logical address and varies from 0000h to FFFFh.

The physical address range occupied by a full segment in Fig. 6 is 32520h to 4251Fh. It is represented in seg:offset form as 3252:0000h to 3252:FFFFh. Observe that in the whole range, segment part is fixed (3252h) whereas offset part varies from 0000h to FFFFh. The segment address is stored in one of the segment registers and the offset part is either stored in register or directly stored in instruction depending upon the addressing mode of the instruction.

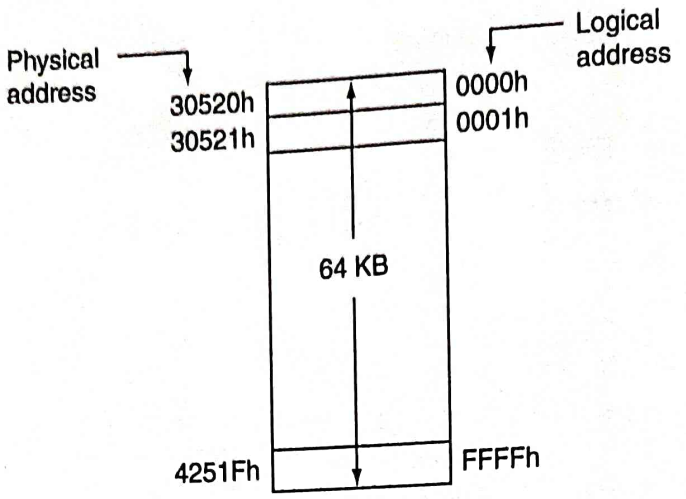


Figure 6 The 8086 segment.

The memory system understands only 20-bit physical addresses. Whenever the 8086 wants to perform read or write operation with the memory, the adder-shifter circuit in the BIU combines the segment and offset parts to generate the 20-bit physical address as follows.

Physical address = Segment address × 16 + Offset address

The segment address is shifted four times left (equal to multiplication by 16) and the offset address is added to compute the physical address. The shifting toward left four times inserts four 0s from right. This means that the segment address contains four lower bits as hidden 0s. Shifting the segment address four times left inserts four 0s in lower positions to make it 20-bit and adding offset to that generates the required physical address. For example, the address 3252:0005h is converted to physical address 32525h as shown in Fig. 7.

This reveals the important fact that we cannot start a segment from any address in the 8086 memory map. The segment can be started from only the addresses having lower four bits 0. Such addresses are 00000h, 00010h, 00020h, ... , FFFF0h which are 16 bytes apart. The 16 bytes distance between them is called *paragraph boundary* which defines the minimum size of the segment. This means that an 8086 segment can have a minimum size of 16 bytes and a maximum size of 64 KB.

The 8086 supports four different types of segments, namely, code, data, stack and extra segments using their segment registers CS, DS, SS and DS, respectively. The segment registers hold the upper

SA×10H + Offset = PA

3252h × 16 → 32520h	Segment address shifted 4 times left
+0005h	Offset address

32525h	Physical address

Figure 7 Computing a physical address.

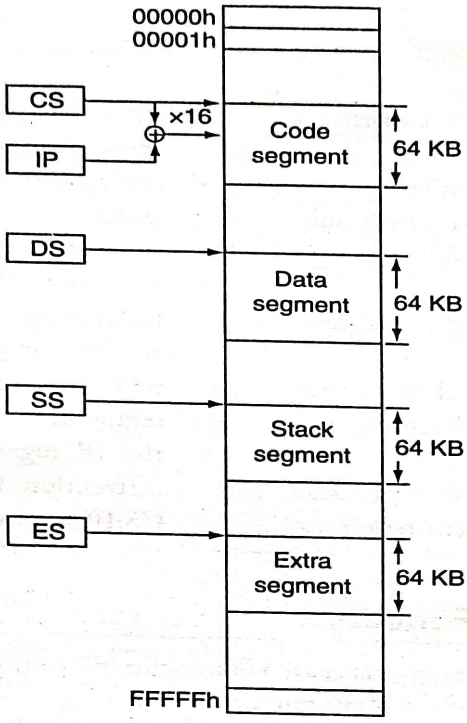


Figure 8 The code, data, stack and extra segments.

16 bits of the starting addresses of the segments. Figure 8 shows the code, data, stack and extra segments pointed by CS, DS, SS and ES, respectively.

The code segment is used to store the program instructions, the data segment is used to store the data, the stack segment provides the stack space for temporary values during the execution and extra segment acts as another data segment. The advantage of segmentation is that the three different components of a program – code, data and stack – reside in separate memory and thus changes in one can be made independent of others. The change in data segment does not change the code or stack segment and vice versa.

The 8086 combines segment base from one of the segment registers with the offset to access the memory. The offset addresses for the different segments are provided differently. The 8086 BIU contains one more 16-bit register called *Instruction Pointer* (IP) as shown in Fig. 8. It is always used as offset address for the code segment register to provide the physical address of the next instruction to be executed and incremented each time by the length of the instruction to point to next instruction in the code segment. This means that CS:IP acts as a Program Counter and provides the address of next instruction to be executed as follows.

Physical address of next instruction = CS × 16 + IP

The offset addresses for the other segment registers are provided by the instruction depending on the addressing mode used by the instruction. The addressing modes are discussed in Chapter 2.

Check Points

We now know that

- the memory map of the 8086 consists of address range 00000h to FFFFFh.
- the 8086 memory is divided into chunks of 64 KB known as segments. At a time only one 64 KB chunk is accessible through one of its segment registers.
- the 8086 contains four segment registers: CS, DS, SS and ES.
- the 20-bit physical address is divided into segment and offset parts and represented as seg:offset format.
- the 8086 computes the 20-bit physical address by shifting the segment base four times left and adding the offset, that is physical address = segment × 16 + offset.
- the segment can start only from the physical address having four lower bits 0. This defines the paragraph boundary of 16 bytes.
- the code segment stores the program instructions, data segment stores the data, the stack segment provides the stack space and extra segment acts as additional data segment.
- the IP register stores the offset of next instruction in code segment and hence CS:IP acts as program counter.

1.7 The 8086 Pin Functions

The 8086 is Intel's first 16-bit microprocessor manufactured as 40 pin DIP (Dual In line Package). Figure 9 shows the pin diagram of the 8086 microprocessor.

The 8086 microprocessor operates in two different modes: minimum mode and maximum mode. The pin MN/$\overline{MX}$ is used to decide the operating mode. If the MN/$\overline{MX}$ pin is tied to the Vcc (Logic 1), then the 8086 operates in the minimum mode and if the MN/$\overline{MX}$ pin is tied to the ground (Logic 0), then the 8086 operates in the maximum mode. The functions of pin numbers 24 to 31 are different in minimum mode and maximum mode. The rest of the pins have common functions in both the modes. The maximum mode pin functions for pin numbers 24 to 31 are listed in Fig. 9 in brackets. The minimum and maximum modes are discussed in the next section.

We know that the 8086 has 20-bit address bus and 16-bit data bus. The 16-bit data bus D_{15}-D_0 is multiplexed with the address lines A_{15}-A_0 and known as the AD_{15}-AD_0. The address lines A_{19}-A_{16} are multiplexed with the status signals S_6-S_3 and known as A_{19}/S_6-A_{16}/S_3. The output signal, ALE (Address Latch Enable), is used to demultiplex the multiplexed address/data and multiplexed address/status bus. When the ALE is high, the multiplex bus contains the address bits and when it is low, the multiplex bus contains data (lower 16 lines) and status bits (upper 4 lines). The falling edge of the ALE signal latches the address bits using external latch and makes the bus free for data transfer. The same is true for multiplex address/status lines. Once the address bits A_{19}-A_{16} are latched, the lines are used to send the status on S_6-S_3 output pins.

The status signals S_6-S_3 are output signals used to show the internal status of the 8086 microprocessor. The status lines S_4S_3 together give which segment register is used to generate the physical address during current cycle. It is given in Table 3. For example, $S_4S_3 = 01$ shows that the SS register is used as segment base to generate the physical address. The status signal S_5 shows the status of the internal interrupt flag and the signal S_6 is always 0.

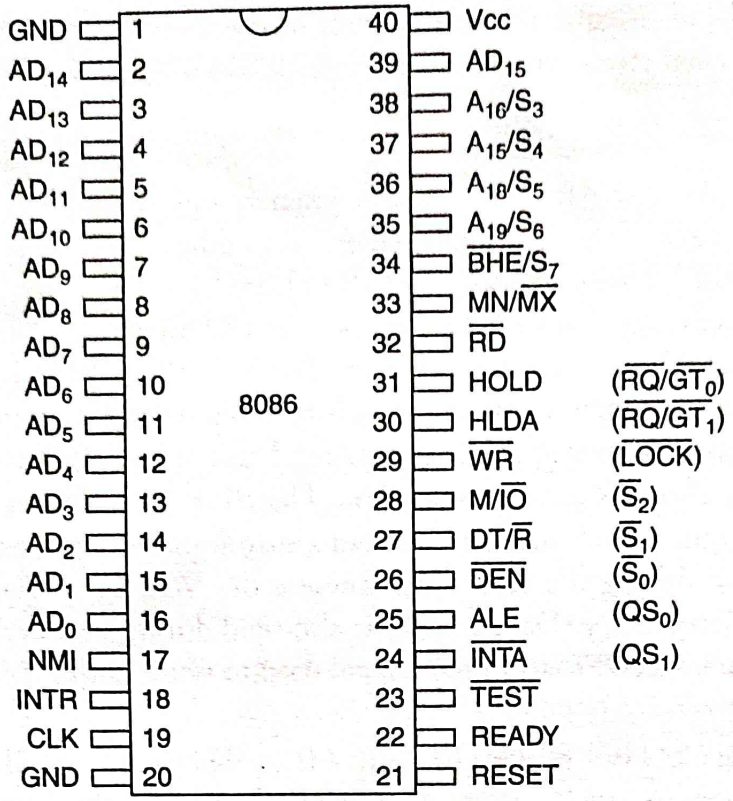


Figure 9 The 8086 pin diagram.

Table 3 Address status with S_4S_3 signals

Signals		*Segment Register Used*
S_4	S_3	
0	0	ES
0	1	SS
1	0	CS/None (0000h)
1	1	DS

The 8086 microprocessor contains three interrupt signals which include two interrupt input pins INTR and NMI, and an interrupt acknowledgement pin INTA. The INTR is general-purpose interrupt and maskable, that is, it can be disabled. The NMI is non-maskable, that is, it cannot be disabled. The INTA is the output signal used to acknowledge the acceptance of interrupt on INTR or NMI pin sent by the external device. The interrupts are covered in detail in Chapter 7.

The TEST and RESET pins are the input signals. The TEST is used to synchronize the 8086 with the external hardware when the 8086 executes the WAIT instruction whereas the 0 on the RESET causes the system reset for the 8086 microprocessor. The HOLD and HLDA pins (minimum mode) are used to interface the 8086 with the DMA (Direct Memory Access) controller. The HOLD is an input signal used by the external DMA controller chip to send the request to the 8086 microprocessor to grant the control of the system bus. The HLDA is an output pin used by the 8086 microprocessor to inform the DMA controller to grant the request. The HOLD and HLDA signals

are replaced by the $\overline{RQ}/\overline{GT}$ (Request/Grant) signals in the maximum mode. The CLK is an input signal connected to external clock circuit. All the operations in the 8086 based system are synchronized to a central clock.

The 8086 provides the various control signals to interface with the memory and I/O devices. We have already understood the ALE signal which when at logic 1 indicates the valid address on the address bus. The falling edge of the ALE signal latches the address into the external latch. The output signal $M/\overline{IO}$ distinguishes between memory and I/O operations. The logic 1 on $M/\overline{IO}$ indicates the memory operation whereas logic 0 on $M/\overline{IO}$ indicates the I/O operation. The DT/$\overline{R}$ (Data Transmit/Receive) is an output signal indicating the direction of data transfer. When it is 1, data is transmitted from the microprocessor and when it is 0, the data is received by the microprocessor. The $\overline{BHE}$ (Bus High Enable) signal is used to enable the odd memory bank in the 8086 memory banks (discussed later in this chapter). It also acts as status signal S_7. The $\overline{BHE}/S_7$ is also an output signal. The $\overline{RD}$ and $\overline{WR}$ signals are output signals and acts as read control and write control signals, respectively. The $\overline{RD}$ signal goes low during the read cycle whereas the $\overline{WR}$ signal goes low during the write cycle. Another output signal $\overline{DEN}$ (Data Enable) is also send during read cycle to enable the external device to provide the data. READY is an input signal used to synchronize the slower devices with the microprocessor.

The pins 24 to 31 act as HOLD, HLDA, $\overline{WR}$, $M/\overline{IO}$, DT/$\overline{R}$, $\overline{DEN}$, ALE and $\overline{INTA}$ signals in minimum mode. They act as $\overline{RQ}/\overline{GT}_0$, $\overline{RQ}/\overline{GT}_1$, $\overline{LOCK}$, $\overline{S}_2$, $\overline{S}_1$, $\overline{S}_0$, QS_0 and QS_1 signals in maximum mode.

Check Points

We now know that

- the 8086 is a 40 pin DIP chip.
- the 8086 works in minimum and maximum modes. It is set either in minimum or maximum mode using $MN/\overline{MX}$ pin.
- the 8086 provides the 20-bit address bus and 16-bit data bus. The data bus is multiplexed with the lower 16 address lines and is known as AD_{15}-AD_0.
- the upper 4 address lines A_{19}-A_{16} are multiplexed with the status signals S_6-S_3.
- the ALE signal is used to demultiplex the multiplexed address/data and address/status signals.
- the INTR and NMI are the interrupt input pins and the $\overline{INTA}$ is interrupt acknowledgement pin.
- the $\overline{TEST}$ is used for synchronization during WAIT instruction and RESET provides the system reset.
- HOLD and HLDA signals are used for interfacing with the DMA controller.
- the external clock circuit provides the clock signal to the CLK signal.
- the control signals used to interface with memory and I/O include the ALE, $M/\overline{IO}$, DT/$\overline{R}$, $\overline{BHE}/S_7$, $\overline{RD}$, $\overline{WR}$, $\overline{DEN}$ and READY.
- the pin numbers 24 to 31 have different functions in minimum and maximum modes.

1.8 Minimum and Maximum Mode

We know that the 8086 microprocessor operates in two different operating modes: minimum mode and maximum mode. The minimum mode is used for simple system having a single processor whereas the maximum mode is used for complex system with multiple processors. These two modes enable the system designer to use the 8086 microprocessor for designing simple and small systems to complex and large systems. Let us take the overview of the minimum and maximum modes.

Minimum Mode

When the pin MN/$\overline{MX}$ is tied to the logic 1, the 8086 operates in the minimum mode. The 8086 generates all the necessary control signals to interface itself with the memory and I/O devices directly (as shown in Fig. 10) on the pins 24 to 31. The meanings of these signals are already discussed in previous section.

Maximum Mode

When the pin MN/$\overline{MX}$ is tied to the logic 0, the 8086 operates in the maximum mode. The pins 24 to 31 play different roles in the maximum mode. The HOLD and HLDA signals of minimum mode are now replaced by the $\overline{RQ}/\overline{GT}$ (Request/Grant) signals; $\overline{WR}$ by the $\overline{LOCK}$ signal; M/$\overline{IO}$, DT/$\overline{R}$ and $\overline{DEN}$ by $\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$ signals; and ALE and $\overline{INTA}$ by the QS_0 and QS_1 (Queue Status) signals. Figure 11 shows the maximum mode 8086 microprocessor.

As can be seen from the figure, the 8086 does not provide the control signals for interfacing with memory and I/O devices directly; rather it provides the status signals $\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$. These status signals are used as input to the external bus controller chip 8288. The 8288 generates the necessary control signals based on the status signals. Further discussion of the maximum mode system is beyond the scope of this book.

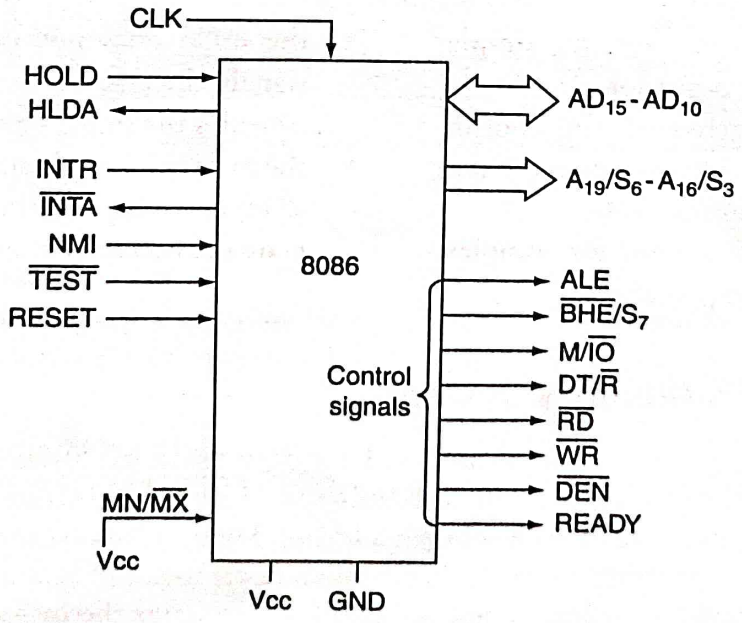


Figure 10 The minimum mode 8086 microprocessor.

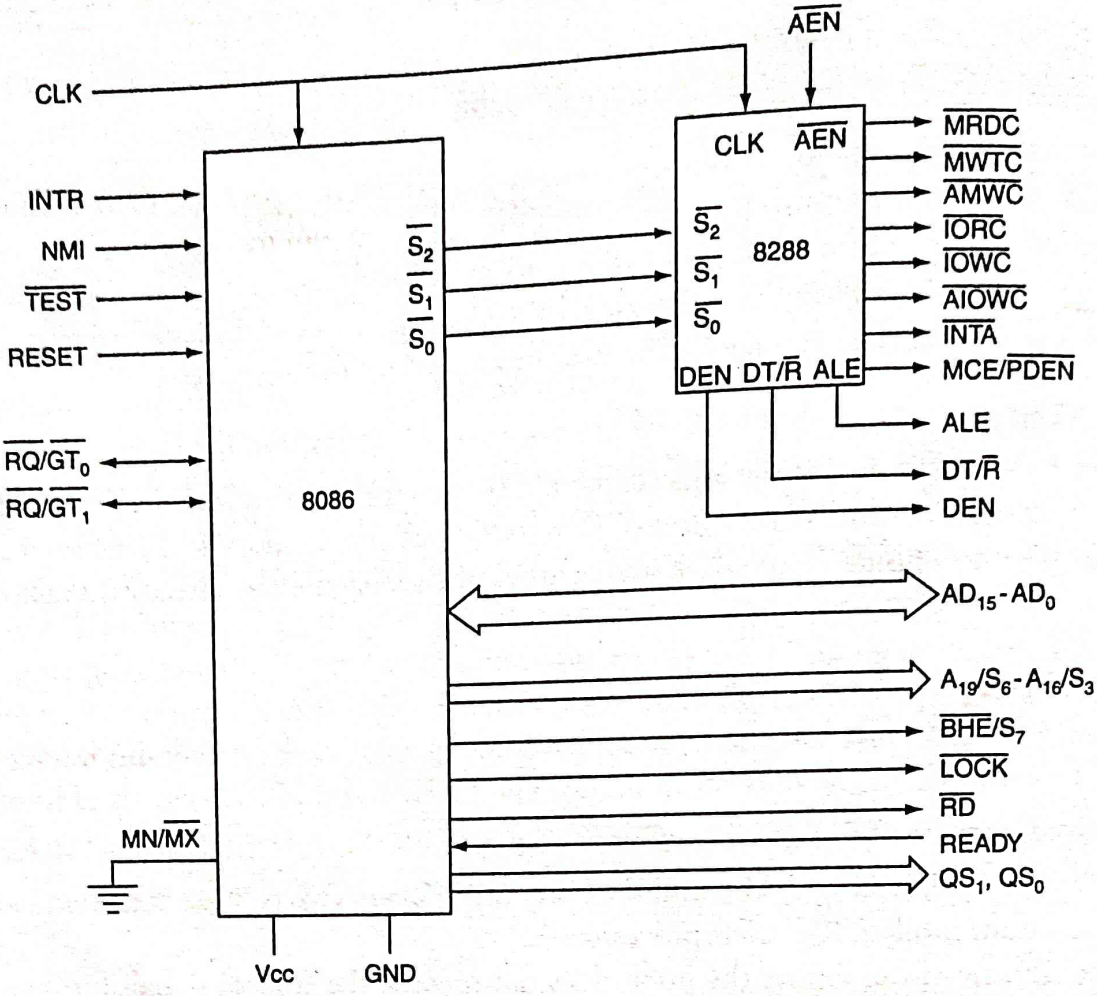


Figure 11 The maximum mode 8086 microprocessor.

Check Points

We now know that

- the minimum mode is used for simple system having single processor.
- the 8086 directly generates the control signals necessary to interface memory and I/O devices in minimum mode.
- the maximum mode is used for complex system with multiple processors.
- the 8086 does not generate the control signals directly in maximum mode, but provides the status signals.
- the maximum mode status signals are given as input to the 8288 bus controller to generate the necessary control signals.

1.9 The 8086 Memory System

The 8086 microprocessor can address total 1 MB memory using its 20-bit address bus. The 8086 memory is organized into two banks in order to utilize the 16-bit data bus to transfer the 16-bit data (word) at a time (when the word starts from even address). Figure 12 shows the 8086 memory organization as the odd and even banks. The even bank contains the memory locations with even addresses 00000h, 00002h, 00004h, ..., FFFFEh and the odd bank contains the memory locations with odd addresses 00001h, 00003h, 00005h, ..., FFFFFh. The lower order data bus D_7-D_0 is connected to even

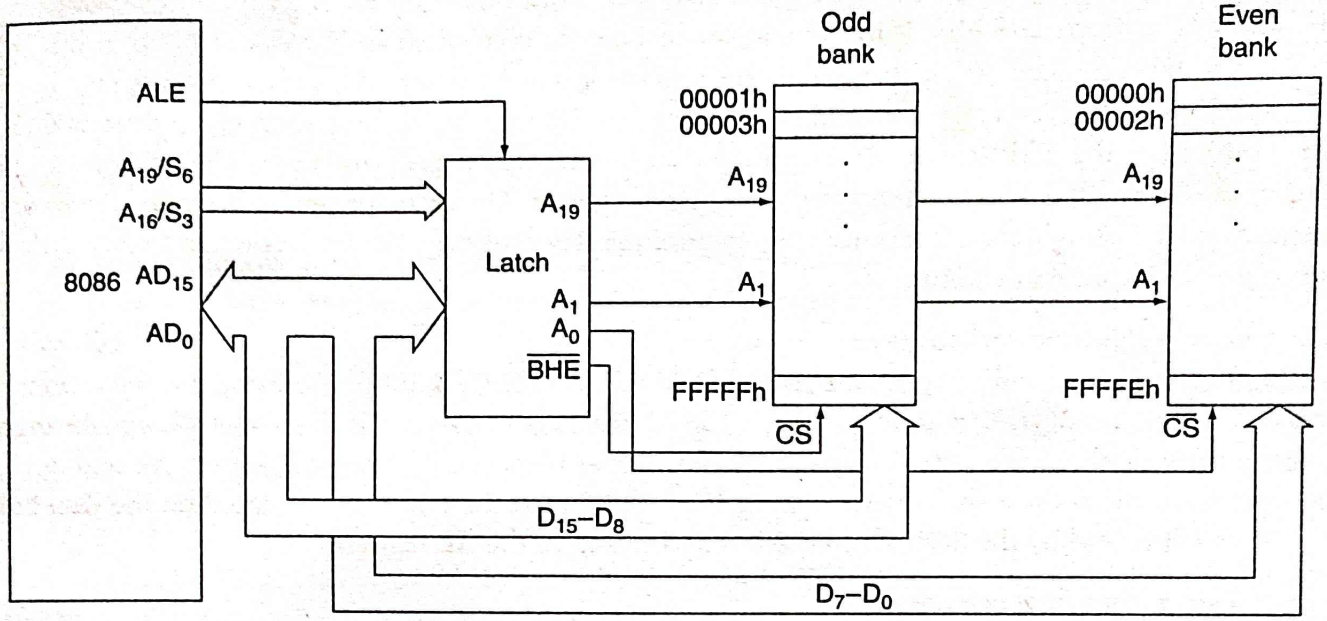


Figure 12 The 8086 memory banks.

bank and the higher order data bus D_{15}-D_8 is connected to the odd bank. The address line A_0 is used to enable the even bank (0 = enable, 1 = disable). The 8086 microprocessor has a special signal $\overline{BHE}$ (Bus High Enable) which is used to enable the odd bank (0 = enable, 1 = disable). The address lines A_{19}-A_1 are connected to both the banks and used to address the specific location in the banks when a bank is enabled. Remember that the actually the data bus D_{15}-D_0 is multiplexed with the address lines A_{15}-A_0 and is known as AD_{15}-AD_0 in the 8086 pin functions. The ALE signal is used to demultiplex the AD_{15}-AD_0 into data bus D_{15}-D_0 and address lines A_{15}-A_0 using the latch. The signals shown in Fig. 12 are demultiplexed signals.

The 8086 supports both the 8-bit (byte) and 16-bit (word) data transfers in both the directions. Let us understand how we can perform read or write operation either for a byte or a word. We will use the read operation to explain the working of the 8086 memory banks. The functioning of memory banks remains same for write operation except the direction of data transfer. There are four possible cases: read a byte from even address, read a word from even address, read a byte from odd address and read a word from odd address. They are explained as follows.

Case 1: Read a byte from even address

Assume that we are executing an instruction MOV AL, BYTE PTR [0000h]. It copies a byte from a physical address DS:0000h (even address) into the AL register. The $A_0 = 0$ in the address enables the even memory bank. The 8086 inserts the $\overline{BHE} = 1$ at the same time which keeps the odd bank disabled. The byte from the address 00000h (assuming DS = 0000h) in the even bank is placed on the data bus D_7-D_0 which is read by the 8086 from the bus and stored in the AL register.

Case 2: Read a word from even address

Assume that we are executing an instruction MOV AX, WORD PTR [0000h]. It copies the lower byte of the word from the physical address DS:0000h into the AL register and higher byte of the word from the physical address DS:0001h into the AH register. Remember that a word is stored in two consecutive

locations with lower byte followed by a higher byte. The word boundary is even here as the word starts from an even address. The $A_0 = 0$ in the address enables the even memory bank. The 8086 inserts the $\overline{BHE} = 0$ at the same time which also enables the odd bank. The byte from the address 00000h (assuming DS = 0000h) in the even bank is placed on the data bus D_7-D_0 and the byte from the address 00001h (DS = 0000h) in the odd bank is placed on the data bus D_{15}-D_8 at the same time. The 8086 reads the word from the data bus D_{15}-D_0 and stores into the AX register. The entire operation is completed in one machine cycle. The user should take care to organize the words at even boundaries to get higher performance specifically arrays of words.

Case 3: Read a byte from odd address

Assume that we are executing an instruction MOV AL, BYTE PTR [0001h]. It copies a byte from a physical address DS:0001h (odd address) into the AL register. The $A_0 = 1$ in the address keeps the even memory bank disabled. The 8086 inserts the $\overline{BHE} = 0$ at the same time which enables the odd bank. The byte from the address 00001h (assuming DS = 0000h) in the odd bank is placed on the data bus D_{15}-D_8 which is read by the 8086 from the bus and stored in the AL register.

Case 4: Read a word from odd address

Assume that we are executing an instruction MOV AX, WORD PTR [0001h]. It copies the lower byte of the word from the physical address DS:0001h into the AL register and higher byte of the word from the physical address DS:0002h into the AH register The word boundary is odd here as the word starts from odd address. During the first machine cycle the address placed on address bus is DS:0001h. The $A_0 = 1$ in the address keeps the even memory bank disabled. The 8086 inserts the $\overline{BHE} = 0$ at the same time which enables the odd bank. The byte from the address 00001h (assuming DS = 0000h) in the odd bank is placed on the data bus D_{15}-D_8. The 8086 reads the byte from the data bus D_{15}-D_8 and stores into the AL register. During the second machine cycle, the address placed on address bus is DS:0002h. The $A_0 = 0$ in the address enables the even memory bank. The 8086 inserts the $\overline{BHE} = 1$ at the same time which keeps odd bank disabled. The byte from the address 00002h (assuming DS = 0000h) in the even bank is placed on the data bus D_7-D_0. The 8086 reads the byte from the data bus D_7-D_0 and stores into the AH register. Thus, the first machine cycle reads the lower byte from the odd bank and the second machine cycle reads the higher byte from the even bank. Thus, the word at the odd address needs two cycles to read.

Table 4 summarizes the above cases with status of A_0 and $\overline{BHE}$ signals, numbers of cycles needed and the data lines involved in the operation.

Table 4 Status of signals during read operation

Case	*Signals*		*No. of Machine Cycles*	*Data Bus*
	A_0	$\overline{BHE}$		
Read a byte from even address	0	1	1	D_7-D_0
Read a word from even address	0	0	1	D_{15}-D_0
Read a byte from odd address	1	0	1	D_{15}-D_8
Read a word from odd address	1	0	2	D_{15}-D_8
	0	1		D_7-D_0

Check Points

We now know that

- the 1 MB memory of the 8086 microprocessor is organized into two banks: odd and even bank.
- the even bank contains all the even addresses whereas the odd bank contains all the odd addresses.
- the even bank is connected to lower order data bus D_7-D_0 and the odd bank is connected to higher order data bus D_{15}-D_8.
- the signal A_0 is used to enable the even bank and the signal $\overline{BHE}$ is used to enable the odd bank.
- the 8086 can read or write 8-bit or 16-bit data. It needs only one memory cycle to read or write 16-bit data starting from even address. For, the 16-bit data starting from odd address it needs two cycles.

Summary

- Intel's journey of microprocessor started with 4-bit microprocessor 4004 and resulted into 80X86 series of microprocessors including 8086/88, 80286, 80386, 80486 and Pentium series.
- Microprocessor is an integrated chip responsible for all the computations and internally consists of three major parts: register array, ALU, and control and timing sections. Microprocessor connects to external world using its data, address and control buses.
- Microcomputer is a digital system consisting of microprocessor, memory and I/O devices connected using system bus. The microprocessor acts as a central device in the microcomputer system.
- The 8086 is an Intel 16-bit microprocessor having 16-bit data bus and 20-bit address bus addressing total of 1 MB memory.
- The 8086 architecture is internally divided into two hardware sections: BIU and EU. The BIU is responsible for external interface and the EU is responsible for execution of instructions.
- The BIU and EU work in parallel. While EU is busy with the execution of instruction, the BIU fetches the next instruction and puts it into the prefetch queue. The prefetch queue can store 6 instruction bytes.
- The BIU contains prefetch queue, four segment registers and an Instruction Pointer whereas the EU contains ALU, general-purpose registers, pointer registers, index registers and a flag register.
- The 8086 memory map consists of address range 00000h to FFFFFh. The 1 MB memory is divided into segments of 64 KB and at a time only one of the segments is accessible through one of its four segment registers.
- The 20-bit physical address is divided into two 16-bit parts: segment base and offset. The segment base is the starting address of the segment and offset address is the distance from the segment base.
- The 8086 computes the physical address by shifting segment address four times left and adding the offset.

- The code segment stores the program instructions, the data segment stores the data, the stack segment provides the stack space and the extra segment acts as another data segment.
- The CS:IP acts as program counter and contains the physical address of the next instruction to be executed.
- The 8086 is a 40 pin DIP chip. The 8086 pin functions include address bus, data bus, status signals, interrupt signals, control signals and other signals.
- The 8086 operates in two different modes: minimum mode and maximum mode. The mode is decided by the MN/$\overline{MX}$ pin.
- The minimum mode generates the control signals directly whereas the maximum mode uses bus controller 8288 to generate the control signals.
- The 8086 memory is organized into even and odd memory banks with even bank consisting of even addresses and odd bank consisting of odd addresses. The A_0 and $\overline{BHE}$ signals are used to enable the even and odd banks, respectively.

Glossary

Microprocessor is an integrated chip responsible for all the computations including arithmetic and logical operations.

Microcomputer is a digital system consisting of microprocessor, memory and I/O devices connected using system bus.

Bus is a group of signals or wires.

Data bus is a bidirectional bus used to transfer data between microprocessor and memory or I/O devices.

Address bus is a unidirectional bus used to carry the memory or I/O address during read and write operations.

Control bus consists of various individual signals and group of signals used to synchronize the activities with the external devices.

System bus connects the microprocessor with memory and I/O devices and consists of data, address and control buses.

ALU stands for Arithmetic and Logical Unit.

BIU stands for Bus Interface Unit.

EU stands for Execution Unit.

Prefetch queue is an instruction queue of 6 bytes in the BIU of the 8086 and used to store the instruction bytes fetched ahead of time.

Segment is a chunk of memory in 8086 memory map with maximum size of 64 KB.

Memory map of the 8086 consists of the address range 00000h to FFFFFh.

Code segment stores the program instructions.

Data segment stores the program data.

Stack segment provides the stack space for program execution.

Extra segment acts as another data segment.

Instruction pointer is a 16-bit register that provides the offset of an instruction into the code segment.

Segment address is an upper 16 bits of the starting physical address of the segment.

Offset address is 16-bit value showing the distance or displacement of from the segment base.

Objective Questions

State whether true/false. Give reason for your answer

1. The 8086 is internally and externally both 16-bit microprocessor.
2. The 8088 contains 6 byte prefetch queue.
3. The program written for 8088 microprocessor cannot run on the 8086 processor.
4. The maximum size of a segment is 64 KB.
5. The registers provide internal memory to a microprocessor.
6. The 8086 contains 4 byte instruction queue.
7. The interrupt flag enables or disables the maskable interrupts on INTR pin.
8. The minimum size of an 8086 segment is 1 byte.
9. The IP register can be used to provide offset address with any of the segment registers.
10. The control signals are generated directly in maximum mode.
11. The 8086 minimum mode is suitable for designing single processor system.
12. The 8288 is a bus controller chip used in the 8086 maximum mode system.
13. The 8086 always needs two machine cycles to read or write the word.
14. The size of even memory bank is 512 KB.
15. The odd bank is enabled when $\overline{BHE}$ signal is 1.

Multiple-Choice Questions

1. Which of the following is a 16-bit microprocessor?
   a. 8008
   b. 8080
   c. 8085
   d. 8088
2. Which of the following sections of the microprocessor provides synchronization with the external devices?
   a. Registers
   b. ALU
   c. Control and Timing
   d. None of the above
3. How much memory does the 8086 address?
   a. 16 KB
   b. 64 KB
   c. 1 MB
   d. None of the above
4. The 8086 contains _____ control flags and _____ conditional flags.
   a. 3, 6
   b. 6, 3
   c. 6, 6
   d. None of the above
5. What is true about the 8086 processor?
   a. Contains 4 bytes prefetch queue
   b. BIU and EU works in parallel
   c. BIU does not contain any registers
   d. All of the above
6. Which of the following is not a function of the BIU?
   a. Instruction fetching
   b. Instruction execution
   c. Address generation
   d. Data read/write

7. Which of the following is true for segment base?
   a. It is a starting address of segment
   b. It is any valid 8086 address
   c. It is stored in any register
   d. All of the above

8. The minimum size of an 8086 segment is
   a. 8 bytes
   b. 16 bytes
   c. 32 bytes
   d. None of the above

9. Which of the following status signals give the status of internal interrupt flag?
   a. S_3
   b. S_4
   c. S_5
   d. S_6

10. Which of the following signals is used to demultiplex the multiplexed address/data bus?
    a. $\overline{\text{TEST}}$
    b. ALE
    c. RESET
    d. None of the above

11. What is true about the minimum mode?
    a. $\text{MN}/\overline{\text{MX}} = 1$
    b. Control signals are directly generated
    c. Used to design the simple system
    d. All of the above

12. Which of the following signals decide the direction of data transfer?
    a. $\overline{\text{RD}}$
    b. $\overline{\text{WR}}$
    c. $\overline{\text{DEN}}$
    d. DT/$\overline{\text{R}}$

13. What is true about the 8288 bus controller chip?
    a. Used in minimum mode
    b. Used in maximum mode
    c. Both a and b
    d. None of the above

14. Which of the following operations need two machine cycles?
    a. To read a byte
    b. To write a byte
    c. To write a word to even address
    d. To read a word from odd address

15. Which of the following values for A_0 and $\overline{\text{BHE}}$ enables both the banks at the same time?
    a. 0,0
    b. 0,1
    c. 1,0
    d. 1,1

Review Questions

1. What is a microprocessor? Explain the internal parts of general microprocessor.
2. What is system bus? Give the functions of each of the bus in the system bus.
3. What do you mean by the fetch-decode-execute?
4. Describe the major functions of the microprocessor.
5. Give and explain the block diagram of the microcomputer system.
6. Compare the Intel 4-bit, 8-bit and 16-bit series of processors.
7. List the features of the 8086 microprocessor.
8. Compare the 8086 and 8088 microprocessors.

9. Draw the logical block diagram of the 8086 processor and explain each part in brief.
10. What is BIU? Discuss the functions of the BIU.
11. What is EU? Discuss the functions of the EU.
12. How do BIU and EU make the instruction fetching and execution of the instruction independent using prefetch queue? Explain properly.
13. List the functions of the BIU and EU and give their functional components.
14. Explain the flag register of the 8086 processor with meaning of each flag.
15. What are conditional and control flags? Show their use.
16. How does the prefetch queue help in improving performance?
17. What is a segment? Explain the segmentation in 8086 and list the advantages of having segmentation.
18. What is paragraph boundary? How it is computed?
19. How does 8086 compute 20-bit physical address? Explain with example.
20. Compute the 20-bit physical address for following segment and offset addresses.
    a. Segment = 9078h, offset = 3245h
    b. Segment = 0000h, offset = 0020h
    c. Segment = FFFFh, offset = 000Ah
21. List the 8086 segment registers and show their use.
22. What is IP register? What is its significance?
23. Give the pin diagram of the 8086 microprocessor and briefly explain the pin functions.
24. What are the control signals? Give their meaning and functions.
25. Compare the minimum and maximum modes of the 8086.
26. Draw the 8086 microprocessor in maximum mode and briefly explain it.
27. What are the different status signals in the 8086 microprocessor? List them and give their meaning and role.
28. Explain the 8086 memory organization in terms of odd and even banks.
29. What are the 8086 memory banks? Describe its working with suitable examples.
30. How does 8086 write a word to an odd address? Explain with example.
31. What is $\overline{\text{BHE}}$ signal? Give its role in memory banks.
32. What is the reason to organize the 8086 memory in odd and even banks?

Answers

True or False. Reasons are left as an exercise

1. True
2. False
3. False
4. True
5. True
6. False
7. True
8. False
9. False
10. False
11. True
12. True
13. False
14. True
15. False

Multiple-Choice Questions

1. (d)
2. (c)
3. (c)
4. (a)
5. (b)
6. (b)
7. (a)
8. (b)
9. (c)
10. (b)
11. (d)
12. (d)
13. (b)
14. (d)
15. (a)

2 Basics of the 8086 Programming

LEARNING OBJECTIVES

At the end of this chapter, you will be able to:

- Define, explain and compare machine, assembly and high-level language programming.
- Define and explain the programming model of the 8086 microprocessor.
- Identify the various fields of an instruction format and use them to prepare machine code.
- List and define the various addressing modes.
- List and explain the various functional groups of the 8086 instruction set.

2.1 Introduction

Microprocessor is a programmable device which is programmed through its instruction set to perform some desired tasks. A program is a sequence of instructions arranged in a logical order to carry out the defined job or task. In general, programs are written either using machine codes of the microprocessor known as *machine language programming* or mnemonics known as *assembly language programming* or using higher level programming languages like C, C++, Java, etc. The assembly language programming is the best approach to program the microprocessor. This needs understanding of the internal architecture of the processor and understating of the instructions provided by the processor.

We have already learnt the architecture of the 8086 microprocessor in Chapter 1. Before we start writing programs in the 8086 assembly language, we need to first understand the instruction set. An instruction is a command to perform an operation on specified data to a microprocessor. Instruction can be used in variety of the ways, known as their *addressing modes*. Addressing modes help the user to use instructions in a flexible manner to suit the situations. 8086 instructions are divided into different groups depending on their functions. These include data transfer instructions, arithmetic instructions, logical instructions, branch control instructions, string instructions, interrupt instructions and processor control instructions.

2.2 Programming Languages

A microprocessor can be programmed in three ways using either machine language programming or assembly language programming or high-level language programming. This section discusses each of these methods of programming with their merits and demerits. At the end of the section, we will see that why do we need to use assembly language programming to program the 8086 throughout the book.

Machine Language Programming

Microprocessor has its own machine language which is unique to that microprocessor and not understood by other microprocessors. Machine language of the microprocessor consists of machine codes or binary codes for each instruction provided by the microprocessor. Writing programs using machine or binary codes is known as *machine language programming*. Hence, a machine language program is a sequence of machine or binary codes in logical order. For example, a machine code for transferring the contents of BX register to AX register is

10001001 11011000

The greatest advantage of the machine language programming is that the programs written in machine language are faster because programmers have more control as they directly program the processor using machine codes. Another advantage is that no translation of program is required as it is in machine codes. Program is stored directly in memory and processor stars running it instruction by instruction. The major problem with the machine language is in memorizing the long binary codes which is difficult for any human being. Larger the instruction set, more are the possible codes. For example, the 8086 instructions result into around 20,000 different binary sequences (i.e., machine codes).

Assembly Language Programming

To overcome the problem of memorizing the binary codes, people started using symbolic names for machine codes; these names are known as *mnemonics*. For example, the mnemonic for transferring data from one register to another register is MOV. Hence, the machine code written in previous sections is written now using mnemonic as

MOV AX, BX

Mnemonics use an English word for each machine instruction. Writing programs using mnemonics is known as *assembly language programming*. Thus, an assembly language program consists of sequence of mnemonics.

The major advantage of assembly language programming is the use of user-friendly mnemonics which are easy to remember as they are formed to represent the operations performed by underlying instructions. Assembly language programs are also as fast as machine language programs, as only machine codes are replaced by mnemonics. The major problem is that the processor does not understand any language other than machine language and hence, we need to translate assembly language program to machine language program before passing to processor for execution. There are two ways of translation from assembly to machine language:

1. The first way is to hand code using instruction templates provided by the processor. The drawbacks of hand coding are that it is time consuming, tedious and prone to the errors. We will see the hand coding with example later in this chapter.
2. The second and widely used approach is to use software tool called *assembler* to perform the translation. We will learn how to use assembler in Chapter 3.

Assembly language is used normally when one needs to directly control the hardware or to write the programs that are very close to the hardware. Our objective is to understand the 8086 processor in depth and hence, we will program it using the assembly language. To write the assembly language programs, we will follow the format given in Fig. 1.

Label	*Mnemonic (Opcode)*	*Operands*	*Comments*
	MOV	CX, 05h	;initialize CX register with 5
	MOV	AX, 00h	;initialize AX register with 0
next:	ADD	AX,CX	;add next number to AX
	DEC	CX	;decrement number
	JNZ	next	;if number is not 0, jump to next

Figure 1 Format of an assembly language program.

The format includes four columns, namely, *label*, *mnemonic* or *opcode*, *operands* and *comments*. The label field is used by the jump instructions to denote the address of the target instruction as address is not known at the time of writing a program. It is converted to an address by an assembler at the time of translation. Label field is always ended by colon (:). The second field is the mnemonics of the instruction, for example, MOV, ADD, CMP, etc. It is also known as opcode (operation code) as mnemonic denotes the operation to be performed by an instruction. The third field is called operands as it shows data values on which operation is to be performed. The last field is comments field used to write comments to help make the program more readable. A comment always starts with semicolon (;). First line in Fig. 1 shows the format while rest of the lines give an example of assembly language program written using that format.

High-Level Language Programming

The major issue with both the machine and assembly language programming is the productivity as each instruction denotes only small operation. The large and complex programs are very difficult and time-consuming to write using them. Real-life applications are normally large and complex and are require delivery in a short period. Machine and assembly languages also need significant knowledge about the processor, which is not common to all the people. High-level language programming solves these problems as it uses English words and phrases to form the statements. Each statement in high-level language denotes many operations of assembly or machine language programming. BASIC, C, C++, Java all are examples of high-level programming languages. They are used to write interactive applications for solving real-life problems.

The major advantages of the high-level language programming are productivity, easy to understand and interactive. A processor cannot understand it directly and hence high-level program needs to be translated into machine language before being submitted to the processor for execution. We can either use interpreter or compiler to perform this translation. The interpreter translates one instruction at a time, executes it and repeats the same for all the instructions whereas a compiler translates the whole program at a time and then executes it. Most of the modern programming languages use the compiler for many reasons.

2.3 The 8086 Programming Model

Programming model of a microprocessor consists of its register set. Registers provide temporary memory to store the operands for the instructions. Each register is given a unique name by which we can refer to it in instructions. The knowledge about register set of a processor is very important for a person

who wants to program it. For programming the 8086 processor, we need to know about its register set (discussed in Chapter 1). Here, we will see them from the programmer's point of view. Figure 2 shows the programming model of the 8086 processor. It includes four general purpose registers, two pointer registers (stack and base pointer), two index registers, one flag register, four segment registers and an instruction pointer.

Programming model includes four 16-bit *general-purpose registers*: AX, BX, CX and DX. They are so called because they are normally used as data registers to store the operand values. Each of them can also be used as two independent 8-bit registers. For example, AX can be used as two 8-bit registers AH and AL, BX as BH and BL, CX as CH and CL, and DX as DH and DL. Thus, they are either used as four 16-bit registers or eight 8-bit registers. Apart from using them as data registers, each of them have special functions in the 8086 programming. AX is known as *accumulator* and used as default operand in some word (16-bit) instructions like multiply, divide and I/O instructions. Same is true for AL for byte (8-bit) instructions. BX is known as *base* register and is used mainly as base address in instructions like translate (XLAT). CX is known as *count* register and is used as counter in looping instructions and repeat prefixes. CL is also used as counter in shift and

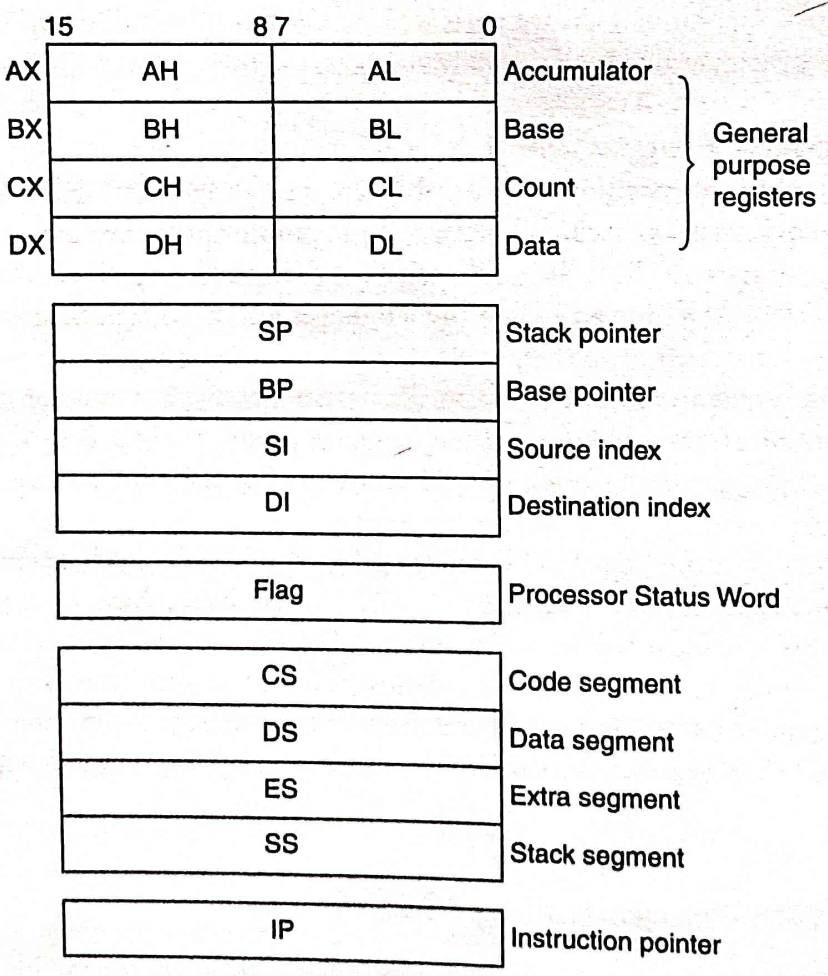


Figure 2 Programming model of the 8086 processor.

rotate instructions. DX is known as *data* register and is mainly used as default operand in multiply, divide and indirect I/O instructions.

Pointer registers are used as only 16-bit registers and cannot be accessed in 8-bit parts. Two pointer registers are called SP (Stack Pointer) and BP (Base Pointer). They are used to store the offset address with respect to segment base stored in SS register. The SP register is used as top of the stack and is incremented or decremented automatically by instructions using stack. BP also contains offset address with respect to segment base in SS and used to access the contents of the stack without changing the top of the stack. It is mainly used to access the parameters passed in subroutine calls using the instructions within subroutine body. We can call them address registers as they store offset and can never be used to store user data.

Two index resisters – SI (Source Index) and DI (Destination Index) registers – are also 16-bit and are used to store the offset address within the data segments and combined with segment address stored in either DS or ES register. They are mainly used with string instructions to hold the offset address within source string and destination string. Normally, index registers are used to store the offset addresses, but we can also use them to store data in some cases such as general-purpose registers. Hence, we can refer to them as *data/address registers.*

The flag register contains nine 1-bit flags including six conditional flags and three control flags which we have already learnt in Chapter 1. Segment registers (CS, DS, ES and SS) and instruction pointer (IP) are already known to you as we have discussed them in detail while learning about segmentation in Chapter 1.

Check Points

We now know that

- microprocessor is programmed using machine, assembly and high-level language programming.
- machine language uses binary codes which are difficult to remember, but machine language programs are faster.
- assembly language uses mnemonics and used for better hardware control. High-level language use phrases of natural language, which provide ease of programming. Both of them need translation to machine codes.
- the 8086 programming model includes its register set including general purpose, pointer and index, segment, instruction pointer and flag registers.

2.4 Instruction Formats

To program a microprocessor, one has to learn its instruction set thoroughly. An instruction acts as a command to the processor to perform an operation on specified data (known as operands). Each instruction contains two parts: *Operation Code* (Opcode) and *Operands*. Operation code is predefined by the processor and fixed for each instruction; however, the processor allows flexibility in specifying operands for the instructions. The flexibility in providing operands for an instruction is known as *addressing modes*. Addressing modes are discussed in detail in this and the next section. For example,

in instruction ADD AX, BX, the mnemonic (Opcode) ADD denotes the addition operation while AX and BX act as operands. In this case, both the operands are in registers, but we can also specify them in memory using different addressing modes.

We need to convert an assembly language instruction into machine code before execution. For this we can either use hand coding or software such as assembler. Let us understand how the hand coding is performed to get the machine codes using examples and at the same time get the idea of how 8086 allows operands to be specified in registers or in memory. The 8086 processor provides a format for each instruction. It is also known as *instruction template*. It contains the fields for operation code as well as operands. We will use MOV instruction as an example for understanding instruction templates, addressing modes and using templates to prepare machine codes.

MOV Instruction: Immediate to Register

Figure 3 shows the format of the "immediate to register" MOV instruction. It is used to load a byte or a word into the register. As shown in the figure, the first byte is divided into three fields: 4-bit OPCODE, 1-bit word size (W) and 3-bit register (REG) fields. The value of OPCODE for this instruction is 1011. If W = 0, the operand is 8-bit (i.e., byte) and if W = 1, the operand is 16-bit (i.e., word). The last three bits specify the binary code for destination register. The binary codes for the 8086 registers are shown in Table 1. The second byte in the template specifies the 8-bit immediate value to be stored in the destination register if W = 0. The third byte is not required if W = 0. Second and third bytes together specify 16-bit immediate value to be stored in destination register with lower byte in second byte and higher byte in third byte if W = 1.

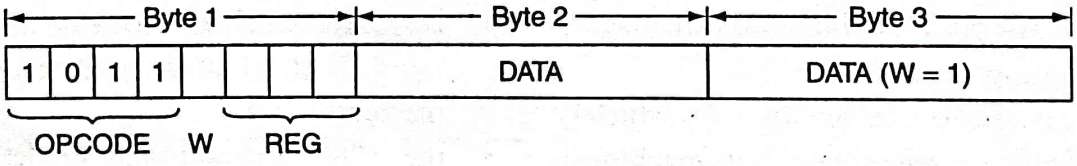


Figure 3 Instruction template for immediate to register MOV instruction.

Table 1 Binary codes for the 8086 registers

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

Figure 4 shows the machine code for MOV AL, 12h (h stands for hex) instruction where first byte contains OPCODE = 1011, W = 0 as operand is 8-bit, REG = 000 the binary code of AL register. The second byte is immediate value 00010010 (12h) to be stored in AL register. Hence, the machine code for instruction MOV AL, 12h is B012 in hexadecimal which will be stored in two consecutive locations in memory [shown in Fig. 6(a)].

The machine code for MOV AX, 1234h is shown in Fig. 5, where the first byte contains OPCODE = 1011, W = 1 as operand is 16-bit, REG = 000 the binary code of AX register. The second byte is the lower byte (34h) of immediate value and the third byte is the higher byte (12h) of the immediate value. Hence, the machine code for instruction MOV AX, 1234h is B83412 in hexadecimal which will be stored in three consecutive locations in memory as shown in Fig. 6(b). Note that 16-bit immediate operand is stored in memory as lower byte followed by higher byte because Intel follows the philosophy of lower byte lower address and higher byte higher address.

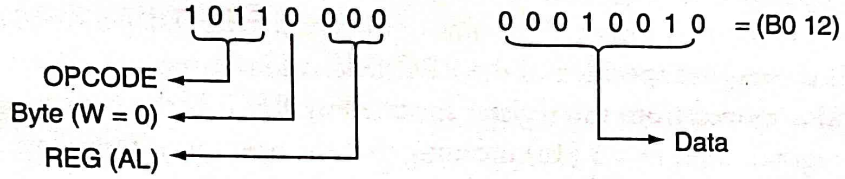


Figure 4 Coding of MOV AL, 12h.

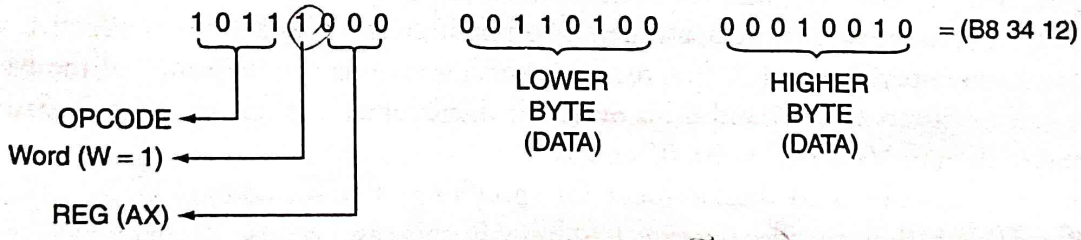


Figure 5 Coding of MOV AL, 1234h.

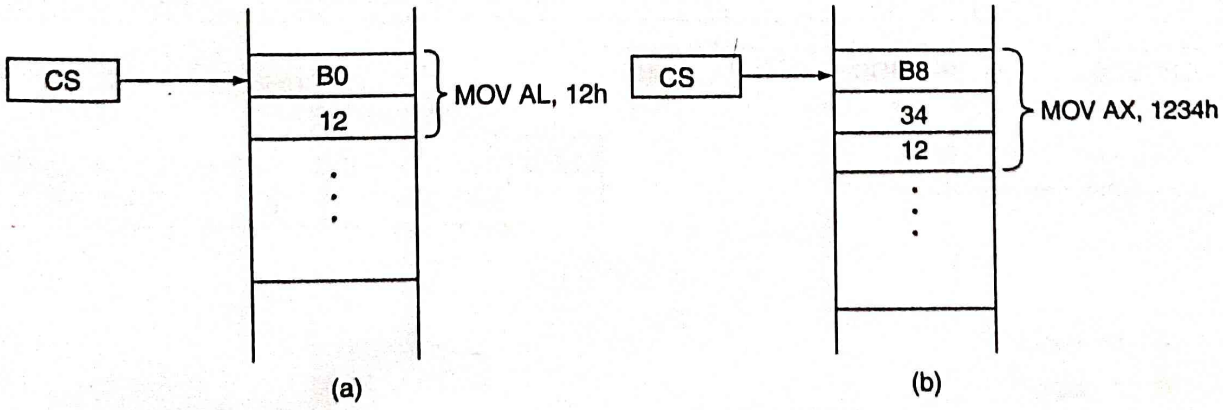


Figure 6 Storage of (a) MOV AL, 12h and (b) MOV AX, 1234h.

MOV Instruction: Between Registers or Between Register and Memory

Figure 7 shows the instruction template of the MOV instruction used to transfer a byte or a word from register to register, register to memory or memory to register. The minimum length of this instruction is 2 bytes and maximum is 4 bytes as shown in the figure. The initial two bytes contain the various fields for opcode and operands, while rest of the bytes contain either displacement value (8-bit or 16-bit) or the direct address depending upon the addressing mode used.

In the figure, the first byte contains 6-bit OPCODE, 1-bit direction D and 1-bit word size W fields. The second byte contains three fields: 2-bit MOD field, 3-bit register field REG and 3-bit register or memory field R/M (Register/Memory). Third and fourth bytes depend on the addressing mode chosen using the fields MOD and R/M. The 3-bit codes for register field REG are already shown in Table 1.

The OPCOCE sequence for this instruction is 100010. The word size field W has the same meaning as immediate to register MOV instruction. If operand to move is a byte then W = 0 and for a word, W = 1. One of the operand in this instruction is always register whereas other is either register or memory location. The REG contains 3-bit code for the register, either source or destination register if both the operands are registers, and register (whether source or destination) if one operand is register and other is memory location. The value of direction field D depends on the register specified by the REG field. If the register specified in the REG field is source register, then D = 0 (From) implying that the operand is moved from the register specified by REG. If the register specified in the REG field is destination register, then D = 1 (To) meaning that the operand is moved to the register specified in the REG field.

MOD and R/M fields together specify one of the 32 addressing modes (24 memory and 8 register) from the table shown in Fig. 8. We know that one of the operand is always register. If the other operand is also register, then MOD = 11 and the R/M contains the 3-bit code for the other register as shown in Fig. 8. If the other operand is memory, then its offset address – also known as effective address – is specified by the memory operand in the instruction using various combinations of the base registers (BX, BP), index registers (SI,DI) and 8-bit or 16-bit displacement. They are given in first three columns corresponding to MOD value 00, 01 and 10.

MOD = 00 does not need displacement for specifying effective address of the other operand. However for R/M=110, it specifies the direct address. In this case, we use alternate format with third

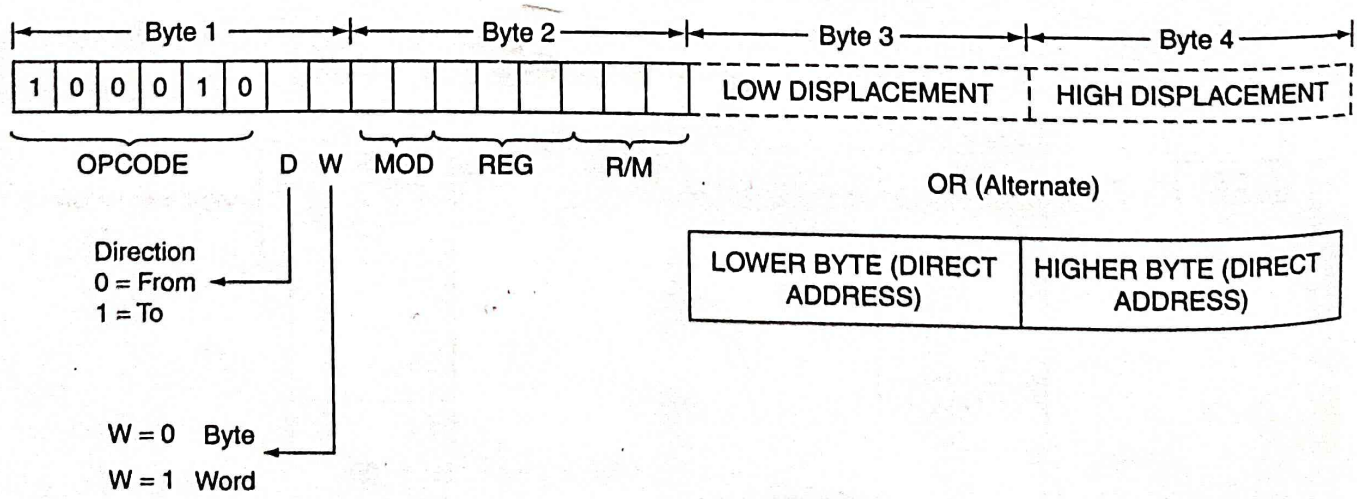


Figure 7 Instruction template for MOV – between registers or between register and memory.

Memory addressing modes → | Register addressing modes →

MOD → / R/M ↓	00	01	10	11 W = 0	11 W = 1
000	[BX] + [SI]	[BX] + [SI] + disp8	[BX] + [SI] + disp16	AL	AX
001	[BX] + [DI]	[BX] + [DI] + disp8	[BX] + [DI] + disp16	CL	CX
010	[BP] + [SI]	[BP] + [SI] + disp8	[BP] + [DI] + disp16	DL	DX
011	[BP] + [DI]	[BP] + [DI] + disp8	[BP] + [DI] + disp16	BL	BX
100	[SI]	[SI] + disp8	[SI] + disp16	AH	SP
101	[DI]	[DI] + disp8	[DI] + disp16	CH	BP
110	data 16 (Direct address)	[BP] + disp8	[BP] + disp16	DH	SI
111	[BX]	[BX] + disp8	[BX] + disp16	BH	DI

disp8 – 8-bit displacement
disp16 – 16-bit displacement

Figure 8 Register and memory addressing modes.

and fourth bytes containing lower and higher bytes of direct address, otherwise first two bytes of the template are sufficient.

MOD = 01 uses the 8-bit displacement in addition to base and/or index registers to specify the effective address of the other operand. In this case third byte is used to store the 8-bit displacement. MOD = 10 uses 16-bit displacement in addition to base and/or index registers to specify the effective address of the other operand. In this case, we use third and fourth byes to store the displacement with lower byte followed by higher byte.

Let us take one example for each case to understand the template and convert the assembly instruction to corresponding machine code.

Example 1

Coding of MOV AX, BX

This instruction transfers the contents of BX register to AX register. This instruction can be interpreted as "to AX register" or "from BX register" for the direction point of view as both the operands are registers. Let us focus on "to AX register". The coding of the instruction is given in Fig. 9.

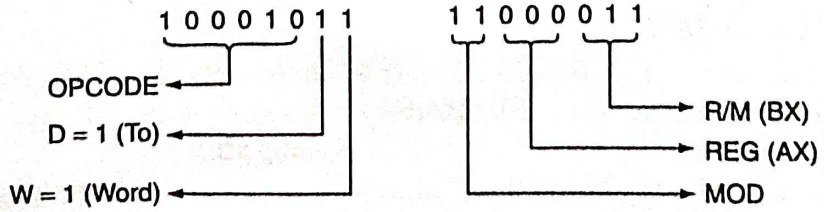


Figure 9 Coding of MOV AX, BX (To).

As can be seen from the figure, OPCODE = 100010, D = 1 as the operand is moved to destination AX register, W = 1 for the word operands, REG = 000 code for AX register, MOD = 11 and R/M = 011 code for other register BX.

If we consider "from BX register", then the coding appears as given in Fig. 10. Observe that D = 0 (From) and REG = 011 (BX) and R/M = 000 (other register AX). This example highlights the important fact that if both the operands are registers, then same instruction can be coded in two different ways.

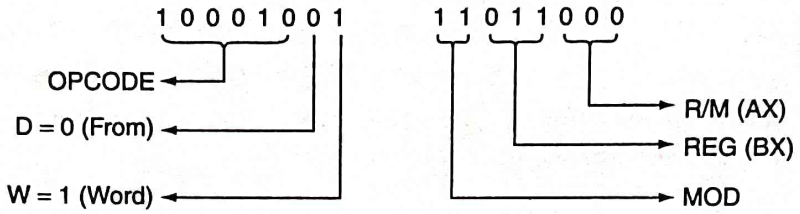


Figure 10 Coding of MOV AX, BX (From).

Example 2

Coding of MOV [SI], CL

This instruction transfers the byte stored in CL register to the memory location with offset (i.e., effective) address stored in SI register in the data segment. In this case, we have only one register which is source and the direction D = 0 (From). The coding of the instruction is given in Fig. 11. As seen in the figure, OPCODE = 100010, D = 0 as transfer from CL register, W = 0 for byte, REG = 001 code for CL register. The other operand is memory location with effective address specified using SI register. Hence, the MOD = 00 and R/M = 100.

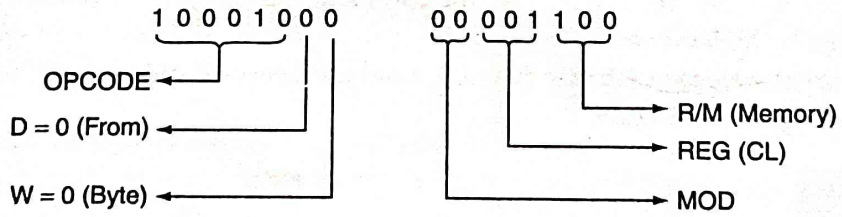


Figure 11 Coding of MOV [SI], CL.

Example 3

Coding of MOV CX, 78h[BP]

This instruction transfers the word from effective address calculated by adding the contents of BP register and 8-bit displacement 78h in the stack segment to CX register. The coding of the instruction is given in Fig. 12. As seen in the figure, MOD = 01, which indicates that the instruction uses 8-bit displacement. The third byte is used to place the 8-bit displacement value 78h.

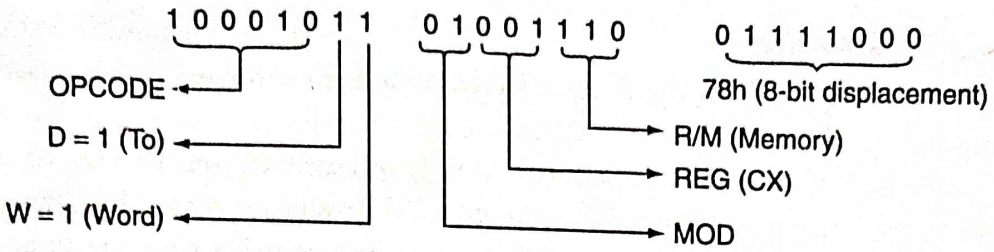


Figure 12 Coding of MOV CX, 78h[BP].

Example 4

Coding of MOV AX, 1234h[BX][SI]

This instruction transfers the word from effective address specified by BX+SI+1234h in the data segment register. The coding of the instruction is given in Fig. 13. As seen in the figure, MOD = 10, which indicates that the instruction uses 16-bit displacement. The third byte is used to place the lower byte of displacement that is 34h and fourth byte is used to place the higher byte of the displacement that is 12h.

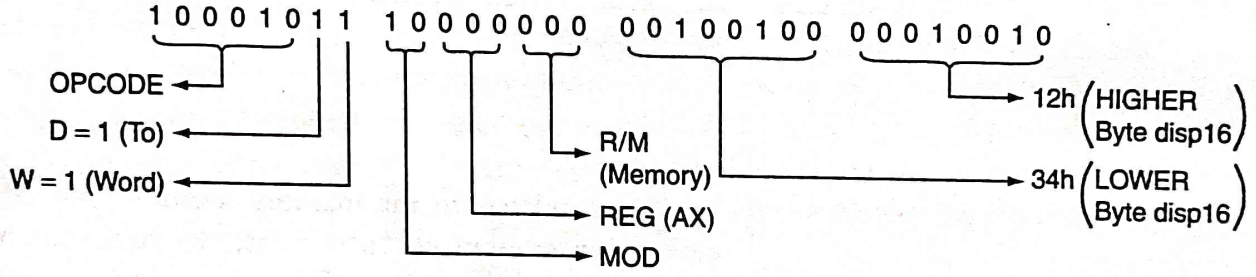


Figure 13 Coding of MOV AX, 1234h[BX][SI].

Example 5

Coding of MOV [3456h], DI

This instruction transfers the word stored in DI register to the effective address 3456h in the data segment. This instruction uses direct address as effective address is provided within instruction. The coding of the instruction is given in Fig. 14. As seen in the figure, MOD = 00 and R/M = 110 indicating direct address. The third and fourth bytes are used to specify the direct address, lower byte followed by higher byte.

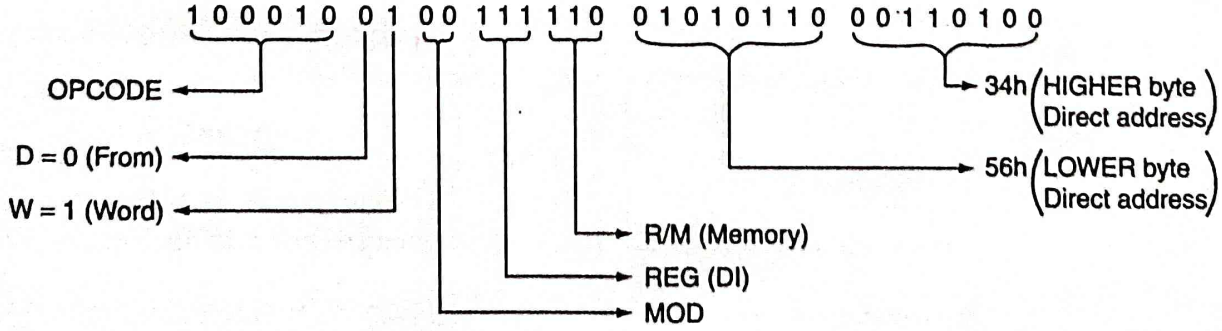


Figure 14 Coding of MOV [3456h], DI.

Segment Override Prefix

For the memory addressing modes in Fig. 8, the effective address is combined with the one of the segment registers to compute the physical address to access the operand stored in memory. The default segment register is stack segment (SS) register if BP is involved fully or partly in computation of effective address; otherwise default segment register is always data segment (DS) register. However, the 8086 provides the way to override this default segment reference using segment override prefix. It is specified using segment register name followed by colon before the specification of memory operand in the instruction. For example,

MOV AL, ES:[DI]

transfers the contents of memory location with offset address specified by DI register in the extra segment (ES) to the AL register. The coding of this instruction needs an extra byte prefix to the machine code of the instruction to indicate the segment. Format for the segment override prefix byte is shown in Fig. 15 with 2-bit segment register codes.

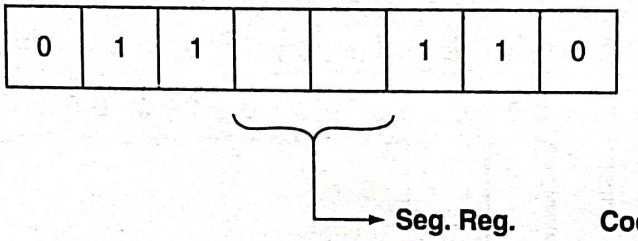


Figure 15 Format of segment override prefix byte.

Example 6

Coding of MOV AL, ES:[DI]

The coding of this instruction is shown in Fig. 16. The first byte is the segment override prefix byte and the rest of the bytes (byte-2 and byte-3) are formed in same manner as previous examples.

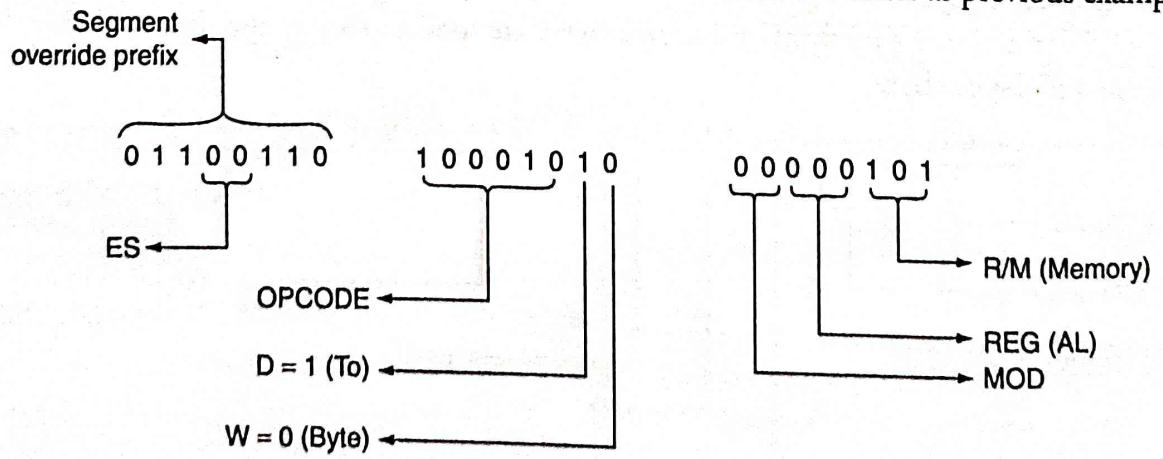


Figure 16 Coding of MOV AL, ES:[DI].

Check Points

We now know that

- an instruction consists of opcode and operands and the 8086 provides instruction template for each instruction.
- immediate MOV contains simple template including opcode, word size, REG and immediate value.
- MOV between registers and register or memory contains complex template and uses 32 addressing modes.
- segment override prefix is used to override the default segment reference.

2.5 Addressing Modes

We have seen in the previous section that operands for the MOV instruction are stored either in registers or in memory or given directly in instruction if it is immediate MOV. Specifying operands in different ways is known as *addressing modes*. More the addressing modes, more flexible the instructions become as same instruction can specify its operands in different manner. Most of the 8086 instructions store their operands in either immediate within instruction, registers or memory. Registers are specified using their unique names provided by the processor, while the effective address of the operand stored in memory specified in number of ways. Figure 8 shows 8 registers addressing modes and 24 memory addressing modes. Some of the instructions in the 8086 processor implicitly specify its operand. In general, we can classify the 8086 addressing modes as

1. Implicit addressing mode.
2. Register addressing mode.
3. Immediate addressing mode.
4. Memory addressing modes:
   - Direct addressing mode.
   - Register indirect addressing mode.
   - Base addressing mode.
   - Index addressing mode.
   - Base Index addressing mode.

We will now discuss these modes in detail with examples to get more clarity.

Implicit Addressing Mode

These types of instructions specify their function and operand both implicitly within the mnemonic itself. For example, the instruction STC is used to set the carry flag CF to 1. In this case, mnemonic STC (Set Carry) specifies the function (Setting CF to 1) and operand (CF from flag register) implicitly and hence it has no explicit operands like other instructions. CLC, CMC, STD are some of the other examples of implicit addressing mode.

Register Addressing Mode

In this addressing mode, the operands of an instruction are stored in specified registers. These types of instructions are faster in execution as operands are stored in internal registers of the 8086 processor and

hence no external operation is required. For byte instructions, operands are stored in 8-bit registers and for word instructions; operands are stored in 16-bit registers. For example, the instruction

MOV AX, CX

moves the contents of CX register to AX register. In this instruction, both source and destination operands are registers. CX is the source operand while AX is the destination operand. More examples of register addressing mode are as follows:

```
MOV AL, BL   ;move contents of BL register to AL register
MOV DS, AX   ;move contents of AX register to DS register
ADD AX, BX   ;add contents of AX and BX reg. and store result into AX
```

Immediate Addressing Mode

In this addressing mode, the operand value is specified in the instruction itself and stored in additional fields provided in instruction format following the opcode bytes. We have studied the immediate to register MOV instruction in the last section, which is example of immediate addressing mode. For example, the instruction

MOV AX, 1234h

moves the value 1234h to AX register. Immediate values are stored in instruction as additional bytes at the end of the instruction. See the instruction format shown in Fig. 3. More examples of immediate addressing mode are as follows:

```
MOV CH, 10h    ;move 10h to CH register
ADD AX, 3425h  ;add 3425h to AX register
AND AL, 0Fh    ;perform logical ADD of AL with 0Fh
```

Memory Addressing Modes

In memory addressing mode, one of the operands must be in the memory. To access the contents in memory, we need to provide the physical address of a memory location where it is stored. The 8086 processor uses 20-bit physical address in the range of 00000h to FFFFFh. The 20-bit physical address is divided into two 16-bit parts: segment base and the offset (or displacement) and represented as **segment:offset**. The segment part shows the base address of the 64 KB segment and is stored in one of the segment register. The offset part shows the displacement or distance from the segment base and is known as *effective address*. Memory addressing mode specifies how an effective address is computed for an operand stored in memory which then combines with the segment address to generate the physical address of operand.

Memory addressing modes specify different ways to compute the effective address of an operand. Considering the ways 8086 specifies the effective address of an operand, there are five memory addressing modes:

1. Direct addressing mode.
2. Register Indirect addressing mode.
3. Base addressing mode.
4. Index addressing mode.
5. Base-Index addressing mode.

They are explained in detail as follows.

Direct Addressing Mode

In direct memory addressing mode, the effective address (offset) is specified directly in instruction just like immediate operand. The physical address is computed as follows:

Physical Address = {segment}:{Direct Address}

Here the default segment register is DS and the Direct Address is 16-bit offset address provided within the instruction. For example, instruction

MOV AX, [0050h]

reads the word from the effective address 0050h (Direct Address) in the data segment, that is, physical address DS:0050h (lower byte from offset 0050h and higher byte from offset 0051h). Remember that [ ] denotes the contents of. Using segment override prefix we can use any of the four segments to get the data. For example, the instruction

MOV AX, ES:[0050h]

refers to the offset 0050h in extra segment instead of data segment. More examples of direct addressing mode are as follows:

```
ADD AL, [0200h]   ;add contents of DS:0200h to AL
MOV [0420h], CX   ;move contents of CX reg. to DS:0420h
```

Register Indirect Addressing Mode

In register indirect addressing mode, the effective address of an operand is indirectly given in one of the registers including base register BX, base pointer register BP or index registers SI and DI. The physical address is computed as follows:

Physical Address = {segment}:{BX or BP or SI or DI}

Here the default segment register is DS if register used to store the effective address is BX, SI or DI and it is SS if the effective address is held by BP. However, using segment override prefix, we can use any segment with any register. For example, the instruction

MOV AX, [BX]

loads the AX with the word from physical address DS:BX (lower byte from DS:BX and higher byte from DS:BX+1). More examples of register indirect addressing mode are as follows:

```
ADD CX, [SI]      ;add word from DS:SI to CX register
MOV ES:[DI], AL   ;move contents of AL to ES:DI
OR CH, [BP]       ;perform logical OR of CH with SS:BP
```

Base Addressing Mode

This addressing mode specifies the effective address of an operand using combination of base register BX or base pointer register BP plus the 8-bit or 16-bit displacement value. The physical address is computed as follows:

Physical Address = {segment}:{BX or BP} + {8-bit or 16-bit displacement}

The effective address (i.e., offset) is calculated by adding the displacement value to the contents of either BX or BP and then it is combined with the segment base stored in DS in case of BX and SS in

case of BP to compute the final physical address. However, using segment override prefix, we can use any segment register as base. Let us understand it by an example. Assume that DS contains 1234h, BX contains 0005h, then the instruction

MOV AX, 80h[BX]

first adds the 0050h and 80h to get the effective address 01D0h and combines it with contents of DS that is 1234h to get the physical address 1234:01D0h. A word is moved to the AX register from this physical address. The assembler allows above instruction to be written in a more convenient form as

MOV AX, [BX+80h]

More examples of base addressing mode are as follows.

```
MOV [BP+2], AL        ;move contents of AL to SS:BP+2
ADD CX, [BX+1234h]    ;add word from DS:BX+1234h to CX
AND ARRAY[BX], AX     ;logical end of AX with DS:BX+ARRAY
```

Observe that in the third instruction shown above, the displacement is given in the form of variable name ARRAY. This type of addressing mode is normally used to access the elements of more complex data structures. For example, we can use it to access the members of structures in C. The base register points to the start of the structure while displacement value holds the distance of member in bytes from the base. By changing the displacement using lengths of the members, we can get the effective address of member by combining the base and displacement value.

Index Addressing Mode

It is very similar to the base addressing mode, except that instead of base registers, the index registers are used with the displacement to compute the effective address. The physical address is computed as follows:

Physical Address = {segment}:{SI or DI} + {8-bit or 16-bit displacement}

Effective address is combined with contents of DS register by default to get the physical address, if segment override prefix is not used. Following are some examples of the index addressing mode:

```
MOV AL, [SI+34h]      ;load AL with byte from DS:SI+34h
ADD [DI+1234h], AX    ;add AX with word in DS:DI+1234h
```

We can use index addressing mode to access the elements into an array where displacement points to the start of the array, while index register points to the individual elements within the array.

Base addressing mode and indexed addressing mode are also some times referred to as register relative addressing mode as they compute the effective address by adding the displacement to the base or index registers.

Base-Index Addressing Mode

The base-index addressing mode uses base register plus index register and 8-bit or 16-bit displacement to get the effective address. The physical address is computed as follows:

Physical Address = {segment}:{BX or BP}+{SI or DI}+{displacement}

Displacement is optional in this addressing mode and if not present, the effective address is calculated by adding contents of base and index registers only. The default segment register is DS if BX is involved in computing effective address and it is SS if BP is involved in computing effective address. Segment override prefix can override these defaults. For example, the instruction

```
MOV AL, [BX+SI]
```

loads a byte from DS:BX+SI to AL register. It can be also written as

```
MOV AL, [BX][SI]
```

Following are some more examples of the base-index addressing mode.

```
ADD CL, [BX+DI+34h]   ;add CL with byte at DS:BX+DI+34h
OR [BP+SI+1234h], AL  ;logical OR of AL and byte at SS: BP+SI+1234h
```

This is used to access the data from highly complex data structures like array of structures in C where the structure contains array of elements like integers as one of the members. Base register is used to point initially to first structure in array and moves from one structure to another by adding total size of the structure. The displacement is used to point to the individual member in current structure element pointed by base. The index register is used to access the individual elements of member of current structure if it is an array like an integer array.

Note: For all the memory addressing modes, remember that when BX, SI or DI contains the full or part of the effective address, the default segment register is DS to get the physical address whereas when BP contains the full or part of the effective address, the default segment register is SS to get the physical address. Using segment override prefix, we can add effective address to any of the four segment registers.

2.6 The 8086 Instruction Set

The instruction set of microprocessor defines the operations that programmers need while developing programs to perform desired task through microprocessor. The 8086 instruction set contains 94 basic instructions which when used with different addressing modes result into more than 20,000 binary codes. The 8086 instructions support both byte (8-bit) and word (16-bit) operations. Intel 8088 is software compatible to the 8086 and has same instruction set as the 8086 processor. The instruction set of the 8086/8088 is divided into various functional groups based on the operations they perform. They are listed as follows:

1. Data transfer instructions.
2. Arithmetic instructions.
3. Logical instructions.
4. Shift and rotate instructions.
5. Transfer or branch control instructions.
6. Looping or iteration control instructions.
7. String instructions.
8. Subroutine and interrupt instructions.
9. Processor control instructions.

Let us take the overview of the functions of each group and instructions included in each of the groups briefly. The detailed study of these instructions and their use in programming is covered in subsequent chapters. Remember that many of the 8086 instructions have more than one mnemonics. They are represented here using '/' as separator. For example, both SHL and SAL represent same instruction – shift left, represented as SHL/SAL.

Data Transfer Instructions

Data transfer instructions are used for transferring data from one place to other, normally between registers, register and memory, and between register and ports. These includes general-purpose data transfer instructions (move, exchange, translate, push and pop), I/O instructions, address transfer instructions (LEA, LDS and LES) and flag transfer instructions (LAHF and SAHF, PUSHF and POPF). They are listed in Table 2 with their functions.

Arithmetic Instructions

Arithmetic instructions are used to perform the arithmetic operations addition, subtraction, multiplication and division. The 8086 allows arithmetic operations on byte and word operands represented in either binary or Binary Coded Decimal (BCD) form. BCD numbers are used to represent decimal numbers using 4-bit BCD code for each decimal digit. For example, 89 is represented as 1000 1001. Arithmetic instructions include five addition instructions, seven subtraction instructions, three multiplication instructions and five division instructions. Remember that arithmetic instructions change the conditional flags based on the result of operation. They are listed in Table with their functions.

Table 2 Data transfer instructions

Instruction	*Function*
MOV	Moves byte or word between registers or register and memory
XCHG	Exchanges bytes or words
XLAT	Translates a byte stored in AL using table with base address in BX
PUSH	Pushes a word on top of the stack
POP	Pops a word from top of the stack
IN	Reads a byte or word from given port to Accumulator
OUT	Writes a byte or word in Accumulator to given port
LEA	Loads offset address of operand into given register
LDS	Loads given and DS registers from memory
LES	Loads given and ES registers from memory
LAHF	Loads AH register with lower byte of the flag register
SAHF	Stores AH register into lower byte of the flag register
PUSHF	Pushes the flag register on top of the stack
POPF	Pops word from the top of the stack to flag register

Table 3 Arithmetic instructions

Instruction	*Function*
ADD	Adds bytes or words
ADC	Adds bytes or words with carry flag
INC	Increments byte or word by 1
AAA	ASCII adjust after addition
DAA	Decimal adjust after addition
SUB	Subtracts a byte or word from byte or word
SBB	Subtracts with borrow
DEC	Decrements a byte or word by 1
NEG	Performs 2's complement of byte or word
CMP	Compares bytes or words
AAS	ASCII adjust after subtraction
DAS	Decimal adjust after subtraction
MUL	Multiplies unsigned bytes or unsigned words
IMUL	Multiplies signed bytes or signed words
AAM	ASCII adjust after multiplication
DIV	Unsigned division, word by byte or double word by word
IDIV	Signed division, word by byte or double word by word
AAD	ASCII adjust after division
CBW	Converts byte to word. Fill the upper byte of word by sign bit of lower byte
CWD	Converts word to double word. Fills the upper word of double word by sign bit of lower word

Logical Instructions

Logical instructions are used to perform the bit-wise logical operations on byte or word operands. They include logical AND, OR, XOR and complement (NOT) plus the TEST instruction to set the flags based on logical ANDing of operands. They are listed in Table 4 with their functions.

Table 4 Logical instructions

Instruction	*Function*
NOT	Performs 1's complement on byte or word
AND	Performs bit-wise logical AND of two bytes or words
OR	Performs bit-wise logical OR of two bytes or words
XOR	Performs bit-wise logical XOR of two bytes or words
TEST	Performs bit-wise logical AND of two bytes or words to update the flags without changing operands

Shift and Rotate Instructions

Shift and rotate instructions are also performing bit-level operations on bits of byte or word operand. Carry flag is also part of the operands in shift and rotate instructions apart from contents of register or memory location specified. They include shift left and shift right instructions and four rotate instructions. They are listed in Table 5 with their functions.

Transfer or Branch Control Instructions

Transfer control instructions are used to transfer the execution control to target instruction within a program (Table 6). They are normally used for implementing branching from one point to other in a program to implement decisions and looping. Hence, they are also referred to as branch control instructions. They include one unconditional and 18 conditional jump instructions. Conditional jump

Table 5 Shift and rotate instructions

Instruction	*Function*
SHL/SAL	Shifts left the bits of byte or word 1 or more times with 0(s) inserted in LSB(s). For each shift MSB goes into CF.
SHR	Shifts right the bits of byte or word 1 or more times with 0(s) inserted in MSB(s). For each shift LSB goes into CF.
SAR	Shifts right the bits of byte or word 1 or more times with MSB copied into MSB. For each shift LSB goes into CF.
ROL	Rotates left the bits of byte or word 1 or more times with MSB copied into LSB and CF for each rotate.
ROR	Rotates right the bits of byte or word 1 or more times with LSB copied into MSB and CF for each rotate.
RCL	Rotates left the bits of byte or word 1 or more times with MSB copied into CF and CF into LSB for each rotate.
RCR	Rotates right the bits of byte or word 1 or more times with LSB copied into CF and CF into MSB for each rotate.

Table 6 Transfer control instructions

Instruction	*Function*
JMP	Jump unconditionally to target instruction
JC	Jump if CF = 1
JNC	Jump if CF = 0
JE/JZ	Jump if equal/Jump if ZF = 1
JNE/JNZ	Jump if not equal/ Jump if ZF = 0
JO	Jump if OF = 1
JNO	Jump if OF = 0
JP/JPE	Jump if PF = 1/Jump if even parity
JNP/JPO	Jump if PF = 0/Jump if odd parity
JS	Jump if SF = 1
JNS	Jump if SF = 0
JA/JNBE	Jump if above/Jump if not below or equal
JAE/JNB	Jump if above or equal/Jump if not below
JB/JNAE	Jump if below/Jump if not above or equal
JBE/JNA	Jump if below or equal/Jump if not above
JG/JNLE	Jump if greater/Jump if not less or equal
JGE/JNL	Jump if greater or equal/Jump if not less
JL/JNGE	Jump if less/Jump if not greater or equal
JLE/JNG	Jump if less or equal/Jump if not greater

instructions include 10 instructions based on the flags, four for unsigned numbers and four for signed numbers. Conditional jump instructions are commonly used after compare instructions. For conditional instructions, the terms above and below are used to refer the magnitude only (i.e., unsigned numbers) whereas terms greater or below are used for signed numbers.

Looping or Iteration Control Instructions

Many times in a program, we need to repeat a block of instructions a specified number of times. Performing all the instructions in the block once is called *iteration*. Looping instructions are used for that purpose. Such instructions are also known as iteration control instructions as they control the number of iterations on the block of instructions. They use CX register as count register to specify the number of iterations. They include three looping instructions and one special instruction JCXZ to skip a block of instructions if count register CX = 0. These instructions are listed in Table 7 with their functions.

String Instructions

Strings are used to represent the non-numeric information and consist of sequence of characters represented in ASCII. String instructions are used to perform various string operations efficiently (Table 8). They include three repeat prefixes and move, compare, load, store and scan operations. Repeat prefixes are used as prefix to the five string operation instructions to repeat them multiple times. String operation instructions work on both bytes and words and have three different mnemonics where 'B' stands for byte and 'W' stands for word.

Table 7 Looping instructions

Instruction	*Function*
LOOP	Loop through a block of instructions until CX = 0
LOOPE/LOOPZ	Loop through a block of instructions while ZF = 1 and CX ≠ 0
LOOPNE/LOOPNZ	Loop through a block of instructions while ZF = 0 and CX ≠ 0
JCXZ	Jump to target instruction if CX = 0

Table 8 String instructions

Instruction	*Function*
REP	Repeats following instruction until CX = 0
REPE/REPZ	Repeats following instruction while ZF = 1 and CX ≠ 0
REPNE/REPNZ	Repeats following instruction while ZF = 0 and CX ≠ 0
MOVS/MOVSB/MOVSW	Moves byte or word from source to destination string
COMPS/COMPSB/COMPSW	Compares bytes or words in source and destination strings
LODS/LODSB/LODSW	Loads string byte to AL or string word to AX
STOS/STOSB/STOSW	Stores byte in AL or word in AX to string
SCAS/SCASB/SCASW	Scans string by comparing byte in AL to string byte or word in AX to string word

Subroutine and Interrupt Instructions

Subroutine and interrupt instructions are used to transfer or return execution control to and from subroutines and Interrupt Service Routines (ISRs). Subroutine instructions include call and return instructions whereas interrupt instructions include interrupt, interrupt on overflow and interrupt return instructions. They are listed in Table 9 with their functions.

Processor Control Instructions

Processor control instructions change in the way a processor operates. They include flag set/reset instructions, no operation instruction and external hardware synchronization instructions (halt, wait, escape and lock). They are listed in Table 10 with their functions.

Table 9 Subroutine and interrupt instructions

Instruction	*Function*
CALL	Transfers control to given subroutine
RET	Returns control back from subroutine
INT	Transfers control to ISR for given interrupt
INTO	Transfers control to ISR of INT 4 if OF = 1
IRET	Returns control back from ISR

Table 10 Processor control instructions

Instruction	*Function*
STC	Sets carry flag CF = 1
CLC	Clears (or resets) carry flag CF = 0
CMC	Complements carry flag CF = (CF)'
STD	Sets direction flag DF = 1
CLD	Clears direction flag DF = 0
STI	Sets interrupt flag IF = 1
CLI	Clears interrupt flag IF = 0
HLT	Halts the processor until interrupt or reset signal
WAIT	Waits until TEST pin goes low
ESC	Is used to synchronize with external coprocessor like 8087
LOCK	Is used as prefix to instruction. During execution of that instruction, other processor cannot talk to the bus
NOP	No operation. Used for creating delay

Check Points

We now know that

- addressing modes provide the flexibility in specifying the operands for an instruction. The 8086 supports implicit, immediate, register and memory addressing modes. The memory addressing modes include direct, register indirect, base, index, base-index addressing modes.

- the 8086 supports 94 instructions resulting into more than 20,000 binary sequences.
- the 8086 instruction set is functionally divided into various functional groups like data transfer, arithmetic, logic, shift and rotate, transfer control, looping, string, subroutine and interrupt and processor control.

Summary

- There are three ways to program microprocessors: Machine language programming, assembly language programming and high-level language programming.
- Machine language is faster but has disadvantage of memorizing the long binary codes.
- Assembly language uses the mnemonics for binary codes and is used widely to program the processors when more hardware control is required.
- High-level language is interactive, easy to understand and efficient for developing large programs.
- Assembly and high-level language programs need to be translated to machine codes before execution using assembler and complier/interpreter, respectively.
- The 8086 programming model consists of its register set that includes four general-purpose registers, two pointer registers (stack and base pointer), two index registers, one flag register, four segment registers and an instruction pointer.
- An instruction consists of operation code and the operands. The 8086 provides an instruction format or template for each instruction consists of opcode field having fixed binary sequence and fields for specifying byte or word operands.
- The 8086 instructions allow operands for an instruction to be specified in different ways known as *addressing modes.*
- The 8086 addressing modes include implicit, register, immediate and memory addressing modes. The memory addressing modes include direct, register indirect, base, index and base-index addressing mode.
- The 8086 instructions include total of 94 instructions resulting into more than 20,000 binary codes considering addressing modes.
- The 8086 instruction set is divided into nine different groups according to their functions, namely, data transfer, arithmetic, logical, shift and rotate, transfer or branch control, looping or iteration control, string, subroutine and interrupt and processor control instructions group.

Glossary

Machine code is a binary sequence for an instruction understood by processor.

Mnemonic is a symbolic name used to denote a machine instruction also known as opcode.

Machine level language uses binary codes to write the programs.

Assembly level language uses mnemonics or opcodes to write programs.

High-level language uses phrases of natural language to write the programs.

Programming model include register set of a processor.

Instruction is a command to a processor which includes opcode and operands.

Opcode is a operation code for an instruction.

Operands are the data on which operation is to be performed.

Instruction format is a template containing various fields for opcode and operands.

Addressing modes specify the operands in different ways.

Instruction set includes all the instructions supported by a processor.

Instruction group is a set of instructions performing related operations.

Segment override prefix is used to override the default segment reference.

Objective Questions

State whether true/false. Give reason for your answer

1. Machine language programs are slower, but difficult to write.
2. Assembly language program needs translation before execution.
3. Each microprocessor has its own high-level language.
4. Programming model of a processor includes all the registers.
5. SI and DI registers are used for indexing and cannot be used at all as data registers.
6. The 8086 supports only word operands.
7. Memory addressing mode instructions specify effective address of its memory operand always using only base or base pointer registers.
8. Register addressing mode instructions are faster in execution.
9. MOV instruction can move a byte from one memory location to another memory location.
10. Most processor control instructions are implicit addressing mode instructions.
11. The 8086 instruction set does not support instructions for multiplication and division.
12. There are no instructions in the 8086 processor to control the iterations.

Multiple-Choice Questions

1. There are _____ instructions in the 8086 instruction set.
   a. 72
   b. 94
   c. 76
   d. 102
2. Which of the following language is used to write programs to control hardware directly?
   a. Machine language
   b. Assembly language
   c. High-level language
   d. None of the above
3. Which of the following language does not need translation?
   a. Machine language
   b. Assembly language
   c. High-level language
   d. None of the above

4. Which of the following instruction follows index addressing mode?
   a. MOV AX, [BP]
   b. MOV AX, 1234h
   c. MOV 12h[DI], AL
   d. All of the above

5. Which of the following instruction follows register indirect addressing mode?
   a. MOV AX, [SI]
   b. MOV [DI], CL
   c. MOV DX, [BX]
   d. All of the above

6. Which of the following instruction follows the implicit addressing?
   a. ADD
   b. STC
   c. PUSH
   d. LOOP

7. Which of the following field identifies the size of the operands in an instruction?
   a. W
   b. D
   c. REG
   d. R/M

8. The special function of CX register is
   a. Accumulator
   b. Base
   c. Counter
   d. Data

9. Which of the following instruction references the stack segment?
   a. MOV AX, [BP+SI]
   b. MOV AX, [SI]
   c. MOV AX, [BX]
   d. MOV AX, [5634h]

10. Index addressing mode instruction computes the effective address as
    a. {BX or BP} + {displacement}
    b. {SI or DI} + {displacement}
    c. {BX or BP} + {displacement}
    d. {BX or BP} + {SI or DI} + {displacement}

Review Questions

1. Why do you require translation of assembly language program?
2. What is advantage of machine language programming?
3. What is a mnemonic? Give examples.
4. When should you use assembly language programming?
5. What are the benefits of high-level languages?
6. What do you mean by programming model of a processor?
7. Give the format for assembly language program.
8. List the registers in the 8086 programming model.
9. Give the special functions of general-purpose registers.
10. What does opcode specify?
11. What is an instruction? What are the components if an instruction?
12. How does 8086 identify that instruction is 8-bit or 16-bit?
13. Give the instruction format for immediate to register MOV instruction.
14. What do you mean by segment override prefix?
15. What is an addressing mode? List the addressing modes supported by the 8086.
16. What is an effective address of an operand? How can you compute it? Give examples.

17. List the 8086 instruction groups.
18. What are the functions of processor control instructions?
19. Give the machine codes for the following instructions:
    a. MOV CL, 67h
    b. MOV [SI], AX
    c. MOV BX, ES:[1234h]
    d. MOV [SI+12h], CH
    e. MOV SI, [0050h]
20. List the shift and rotate instructions.
21. List the instructions in string group with their meaning.
22. Compare the direct and register indirect addressing modes.
23. Compare machine, assembly and high-level programming languages.
24. Explain the 8086 programming model with diagram.
25. What is an instruction template? Explain it with example.
26. Give the instruction template for MOV instruction to move the data between registers or between register and memory. Clear it with example.
27. Explain any three addressing modes with example.
28. What are the memory addressing modes? Explain any two of them with example.
29. Give the applications of base, index and base-index addressing modes.
30. List the instructions in data transfer group with their functions.
31. What are the looping instructions? Explain them with their meaning.
32. List the functional groups of the 8086 instructions with function of each group in brief.

Answers

True or False. Reasons are left as an exercise

1. False
2. True
3. False
4. True
5. False
6. False
7. False
8. True
9. False
10. True
11. False
12. False

Multiple-Choice Questions

1. (b)
2. (b)
3. (a)
4. (c)
5. (d)
6. (b)
7. (a)
8. (c)
9. (a)
10. (b)

3 Programming with Data Transfer, Arithmetic and Logical Instructions

LEARNING OBJECTIVES

At the end of this chapter, you will be able to:

- List and explain the basic assembler directives.
- Use the basic assembler directives in the 8086 assembly language programs.
- Write and execute the simple assembly language programs.
- Use the "debug" tool to debug the programs and see the results.
- Explain and use the DOS service for character and string input/output.
- Describe the important data transfer instructions and use them in programs.
- Describe the arithmetic instructions and use them in programs.
- Describe the logical instructions and use them in programs.

3.1 Introduction

We have discussed basics of the 8086 programming including its programming model, instruction templates, addressing modes and functional groups of the instructions. One of the ways to program the 8086 processor is to write a program as a logical sequence of instructions and then convert it to machine language using template for each instruction before execution. However, this type of hand coding is very inefficient and error-prone specifically when program size is large. An alternative way is to use a software tool called assembler which translates assembly language program into machine language. This will enable writing and executing programs in an efficient manner.

Assembler is a very powerful tool to write assembly language programs as it provides very useful facilities to programmer such as compliers of high-level languages. To use the assembler for writing 8086 programs, we have to understand these facilities, called *assembler directives*, as well as the structure of a program. Once a program is written and assembled, it can be executed to perform the defined task. Debugger is used to debug the program to find logical errors or to see the results. To write useful programs, we need to explore the instructions of various groups meaningfully. We will use the data transfer, arithmetic and logical instructions to start writing simple programs.

3.2 Basic Assembler Directives

The 8086 assembly language programs consist of two types of statements: the 8086 processor instructions and the assembler directives. The *8086 instructions* are the statements which are converted to machine codes whereas the *assembler directives* are the statements that help the assembler to convert the assembly language program to machine language program correctly and efficiently. The assembler directives direct the assembler

for conversion and are removed once the conversion is done. This implies that for the assembler directives, no machine codes are produced. That is why sometimes assembler directives are also known as *pseudo instructions*. This section describes the basic directives that are required to start writing the 8086 assembly language programs. Additional directives are discussed in subsequent chapters as and when needed.

SEGMENT and ENDS Directives

We know that the 8086 processor supports the segmentation with four different segments: code segment, data segment, extra segment and stack segment. The code segment is used to store the instructions or codes, data and extra segment are used to store the data, and stack segment provides the stack space. Each of the segments occupies the space physically in memory and is known as *physical segment* at the machine level. The assembler needs to provide the facility to create a counter part of the physical segments, that is *logical segments*, to write a program in terms of one or more logical segments. Logical segments are also called *user defined segments*. The assembler provides the SEGMENT and ENDS directive to define the boundaries of a logical segment. The SEGMENT directive indicates the start of new segment and the ENDS directive indicates the end of the segment. Each segment has its own name and SEGMENT and ENDS directives are preceded by the name of the segment. The format of a user-defined segment is as follows:

```
seg_name    SEGMENT
            ...
            ;segment body
            ...
seg_name    ENDS
```

The seg_name is the name of the segment. Segment names are formed just like identifier names in programming languages. The segment body contains statements depending upon what type of segment it is. Code segment contains instructions whereas data segment contains data definition and storage allocation statements. Remember the following important points regarding the segment.

1. Two segments cannot be overlapped. This means that the next segment starts only after the previous segment is closed.
2. Names for the segments are formed using same rules as identifiers in programming languages. Don't use reserve words as names, that is, mnemonics and assembler directives. Similar concept is followed for names of variables, labels, subroutine names, etc.
3. Assembler is case insensitive and SEGMENT and segment are considered as same. Same is true for the instruction mnemonics. We will use capital letters throughout the book for both assembler directives and instruction mnemonics and small letters for identifiers to distinguish them easily.

Example 1

Sample definition of a data segment

```
data    SEGMENT
        n1      DW      1234h
        n2      DW      0A234h
        ans     DW        ?
data    ENDS
```

It defines three 16-bit (i.e. word) variables with names n1, n2 and ans with n1 and n2 are initialized to values 1234h and A234h while ans is not initialized. The name of the segment is data. We can use any valid name for it like _data, data1, data_seg, etc. Observe that it contains only data definitions.

Example 2

Sample definition of a code segment

```
code        SEGMENT
            ASSUME cs:code
start:      MOV AX, 1234h
            MOV BX, 0A324h
            ADD AX, BX
            ...
code        ENDS
```

It defines a logical segment named as code. Observe that it contains only instructions in its body. We can understand it in more details once we discuss the rest of the directives in this section.

ASSUME Directive

We need to tell the assembler about which logical segment corresponds to which type of segment by binding it with the corresponding segment register. In Example 2, the statement

```
ASSUME cs:code
```

informs the assembler that the logical segment with name code is code segment as it is bind to CS register. The general format for the ASSUME statement is as follows.

```
ASSUME seg_reg:seg_name
```

where seg_reg is one of the segment register and seg_name is the name of logical segment. If more than one segment is to be bound, the same syntax is repeated with comma (,) as separator. For example,

```
ASSUME cs:code, ds:data
```

tells the assembler that segment with name code is code segment and segment with name data is data segment. This information is used by the assembler to properly initialize the segment registers with the base of the physical segments created for these segments while loading the program in memory for execution. In the above statement, CS register will point to the first instruction in code segment and DS register will point to the first byte in data segment. Later we will see that when program is loaded, Disk Operating System (DOS) automatically initializes the CS register, while other segment registers DS, ES and SS are to be initialized by the user by writing instructions in beginning of the body of the code segment.

It is possible that same physical segment can be pointed by more than one segment registers, meaning that same physical segment is treated as more than one logical segment. This is specifically the case when we use string instructions because they use data and extra segments to store source and destinations strings. In reality, we use same segment to store both source and destinations. Consider the following statement:

```
ASSUME cs:code, ds:data, es:data
```

This statement indicates that both DS and ES registers point to the same physical segment that corresponds to logical segment with name `data`.

Normally, ASSUME statement is used as the first statement in code segment to bind the segments. However, it is possible to use it in middle of the code segment also to change the segment reference.

END Directive

An assembly language program may contain more than one logical segment written one after another. While assembler is assembling a program, the END directive indicates that there are no more segments and it is the end of the program or module in case a program is divided into modules. The assembler stops once it encounters an END statement. It is used in two different forms:

```
END
```

or

```
END start
```

The first form indicates the end of the program or module while the second form indicates the end of the program or module as well as the entry point of the program for execution. The `start` is a label to an instruction in the code segment from where execution begins.

Keeping all the directives in mind, we can form the general structure of an assembly language program as follows:

```
;define  data segment
data     SEGMENT
         …
         ;define variables and allocate storage
         …
data     ENDS
;define  code segment
code     SEGMENT
         ASSUME cs:code, ds:data
start:   …                    ;entry point
         …
         ;write instructions
         …
code     ENDS
         END start                    ;end of the program
```

The statements that begin with semicolon (;) are comments and only placed for readability. Assembler discards them without any processing.

Data Definition Directives

Data definition directives are used to define the variables and allocate the storage for them. They are normally used to define the variables of various sizes such as 8-, 16-, 32-, 64- and 80-bits. They include the directives for defining a byte (8-bit) called DB, defining a word (16-bit) called DW, defining a double

word (32-bit) called DD, defining a quad word (64-bit) called DQ, and defining a ten bytes (80-bit) called DT. Let us discuss them with proper illustrations.

Define Byte

The directive DB is used to define byte values. It is used to store either 8-bit signed or unsigned integers, or ASCII characters. The directive DB is used to define a single byte or multiple bytes in a sequence. Following examples will give an idea of how DB can be used to define byte variables and allocate the storage for them.

Consider the statements given below.

```
n1    DB   0
n2    DB   45h
n3    DB    ?
```

The first statement defines a variable n1 with initial value 0, the second defines a variable n2 with initial value 45h and the third defines a variable n3 with no initial value which means un-initialized. In all the three cases, memory for a single byte is allocated.

Remember that, the 8086 assembler supports all four number formats: binary, decimal, octal and hex. For binary numbers, letter 'b' or 'B' is used followed by binary digits 0 and 1. The second statement in above case is written using binary number as follows.

```
n2      DB      01000101B
```

To use the octal numbers, letter 'q' or 'Q' is used followed by digits 0 to 7. The same statement is written using octal as

```
n2      DB      105Q
```

and using decimal as

```
n2      DB      69
```

Observe that no special letter follows the digits in case of decimal numbers. The hex numbers use the letter 'h' or 'H' after hex digits to denote the hex numbers. The hex numbers use digits 0 to 9 and A to F (or a to f). We will most of the time use the hex numbers as they are more compact and easy to manage. Note that, whenever a hex number starts with A to F, we have to use 0 preceding it to distinguish it from the variable name. For example, A234h is an invalid number as assembler cannot distinguish it from the variable name. It is identified correctly if is written as 0A234h.

The following statements show the use of DB to define the multiple bytes.

```
a1    DB    10, 20, 30, 40, 50
a2    DB    10 dup(0)
a3    DB    10 dup(?)
a4    DB    10, 20, 5 dup(0)
```

The first statement defines a variable a1 which occupies five consecutive bytes initialized with decimal values 10, 20, 30, 40 and 50. We can consider a1 as array of bytes initialized at the time of defining. The second statement defines a variable a2 which reserves the space of 10 consecutive bytes initialized with 0s. It is same as defining an array of 10 bytes with initial values as 0s. The third state-

ment defines a variable a3 occupying 10 consecutive bytes which are not initialized. The last statement defines a variable a4 occupying seven consecutive bytes with values 10, 20, 0, 0, 0, 0, 0.

We can also define single character or character string using DB. Following examples illustrate them.

```
ch1     DB      'X'
mes     DB      'Hello'
```

The first statement defines a character variable ch1 with value that is equivalent to ASCII value of letter X. The second statement defines a variable mes which stores a sequence of ASCII values of letters H, e, l, l and o. The second statement can also be defined as

```
mes     DB      "Hello"
```

or

```
mes     DB      'H', 'e', 'l', 'l', 'o'
```

Define Word

The directive DW is used to define the word values. It is used to store the 16-bit signed or unsigned integer values. It can define a single word or multiple words in the same manner as define byte, the only difference being that each word occupies two consecutive bytes in memory. Following are some examples showing the use of directive DW:

```
n1      DW      0
n2      DW      0A234h
n3      DW        ?
n4      DW      1234h, 45A2h, 65DBh, 0D23Ch
n5      DW      5 dup(0)
n6      DW      3 dup(5 dup(0))
```

Remember that the 8086 stores a word in two consecutive bytes with lower byte followed by the higher byte. In the second statement, the word 0A234h will be stored in memory as 34h followed by A2h in two consecutive locations. For the fourth statement, the memory storage looks like

```
34, 12, A2, 45, DB, 65, 3C, D2
```

The last statement shows the more complex case. It defines a two-dimensional array of words with 3 rows and 5 columns. It occupies total memory of 30 bytes to store 15 words.

Define Double Word

The directive DD is used to define the double word means 32-bit values. It is used to store either 32-bit signed or unsigned numbers or used to store the 32-bit floating point numbers or long integers. Each double word occupies four consecutive memory locations and is stored from lower byte to upper byte. For example,

```
n1      DW      11223344h
```

defines a double word n1 with value 11223344h. The memory image of this word looks like

```
44, 33, 22, 11
```

as it is stored from lower to higher byte. We can use the directive DD also in the same manner as DB and DW.

Define Quad Word

Directive DQ is used to define a quad word (i.e., 64-bit values). It is used to create variables like double in C programming which defines double precision floating point numbers. A single quad word occupies 8 consecutive bytes in memory and is stored in order of lower byte to higher byte in memory. We can follow the same syntax for defining single or multiple quad words as we used in DB, DW and DD.

Define Ten Bytes

The directive DT is used to define values that need 80-bits (i.e., 8 bytes). It is used to create variables like long double in C programming to store large floating point numbers with higher precision. A single DT occupies 10 consecutive bytes in memory and is stored in order of lower byte to higher byte in memory. We can follow the same syntax for defining single or multiple Ten Bytes as we used in DB, DW and DD.

PTR Directive

It is called pointer directive and very useful while referring to the operands stored in memory. There are three pointer directives, namely, BYTE PTR, WORD PTR and DWORD PTR referring to byte, word and double word from the memory. Consider the following statement that complements the operands at memory location – DS:BX, that is, the offset address contained by BX register in data segment:

```
NEG   [BX]
```

The problem with this statement is that the size of operand is not clear. Whether we should perform operation on byte, word or double word at the location DS:BX. The pointer directive solves this problem by directing the assembler about the size of the operand. Suppose if our intension is to complement a byte at the location DS:BX, the above statement is written as

```
NEG   BYTE PTR [BX]
```

We can use WORD PTR and DWORD PTR to refer the word and double word operands in the same manner. Following are some examples that use the pointer directives:

```
MOV AL, BYTE PTR [SI]
JMP WORD PTR [SI]
```

For the first statement, if we do not use the BYTE PTR, the assembler can find the size from the destination register that it is a byte. However, it is better to specify it as sometimes mistake in specifying other operand may lead to problem. For example, if user writes AX by mistake instead of AL and pointer directive is not used, assembler will refer the word operand. The use of pointer directive results in assembler error of operand mismatch and then the user is prompted to correct it.

SEG and OFFSET Directives

We know that the physical address has two parts called segment address and offset address. For elements of a program like variables or procedures stored in physical memory, we sometimes need to get their segment or offset address or both. The directives SEG and OFFSET are used to get the segment part and offset part of physical address respectively. Consider the following data segment.

```
data  SEGMENT
      n1     DW     1234h
      n2     DW     0A234h
      ans    DW       ?
data  ENDS
```

It defines three word variables n1, n2 and ans. Since all the three variables are part of the same segment, there segment address remains same which is nothing but the contents of DS register, assuming that segment name data is bind to DS. The offset address for all three will be different as the distance of each of them from the base is different. Distance of n1 from base is 0, n2 is 2 and ans is 4. From this we can say that the offset address of n1 is 0000h, of n2 is 0002h and of ans is 0004h. The physical address of the n1 is DS:0000h, n2 is DS:0002h and ans is DS:0004h. The complete physical layout or memory image of the above data segment is shown in Fig. 1.

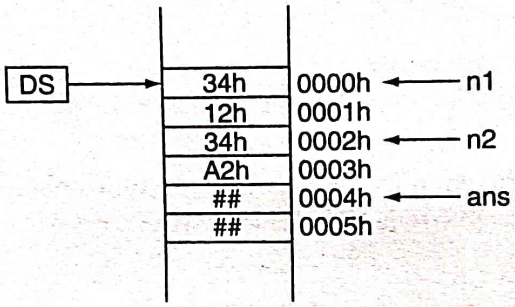


Figure 1 Physical layout of data segment.

The addresses shown in Fig. 1 are known only after loading the program in memory for execution, and not at the time of writing a program. What if we need to use them while writing the programs? We can use SEG and OFFSET directives to direct the assembler to refer the segment and offset addresses by replacing the symbolic references by actual addresses during loading and execution. To get the segment address of variable n1, we can use the following statement:

```
MOV BX, SEG n1          ;get the segment address of n1
```

The above statement loads the BX register with the segment address of variable n1. In our case, it is the contents of DS register as n1 is part of the data segment. Similarly, the following statement gets the offset address of n1 :

```
MOV DX, OFFSET n1       ;get the offset address of n1
```

It loads the offset address of n1 which is 0000h in our case to the DX register. Actually, specification SEG n1 is replaced by contents of DS register and specification OFFSET n1 is replaced by 0000h at the time of loading the program. We will see this when we will discuss the debugging of programs using DOS debugger.

The 8086 assemblers support wide variety of assembler directives for different purposes. We have discussed in this section only those directives which are used very frequently and which are needed to start with the 8086 assembly language programming. We will learn other directives as and when we need them.

Check Points

We now know that

- the assembly program contains the 8086 instructions and assembler directives, also called pseudo-instructions.
- the instructions result into machine codes whereas the directives do not. They simply direct the assembler for correct translation.
- SEGMENT and ENDS directives defines boundaries of logical segments, ASSUME binds the logical segments to segment register and END denotes the end of the program.
- DB, DW, DD, DQ and DT are the directive used to define the variables and allocate the storage.
- PTR is used as byte, word or double word pointer to resolve the conflicts in memory reference.
- SEG and OFFSET are used to get the segment and offset address of variables or procedures.

3.3 Writing and Executing a Program

We are now ready to write a simple assembly language program and see how we can convert it to machine language and execute it. Using a text editor, we have to write our program in a file with extension *.asm*, for example *add.asm*. Assume that we have written code of Program 3.1 in the file *add.asm* to add two 16-bit numbers.

Program 3.1

Write a program to add 16-bit numbers 1234h and 4321h.

Solution

The program is

```
code    SEGMENT
        ASSUME cs:code

start:  MOV AX, 1234h     ;load AX with 1234h
        MOV BX, 4321h     ;load BX with 4321h
        ADD AX, BX        ;add AX and BX

        MOV AX, 4C00h     ;terminate the program
        INT 21h

code    ENDS
        END start
```

The program consists of only one segment named as code which is bind to CS register to direct assembler that it is code segment. The first two MOV instructions initialize the AX and BX registers, respectively. Third instruction adds the contents of AX and BX register and stores the result into AX register. The label `start` to first MOV instruction indicates that the execution begins from this instruction.

When we execute any command or program under DOS (from command prompt in Windows), the control is transferred from operating system to the program. It is necessary that after completion of a program, the control is written back to operating system so that prompt comes back and we can issue the further commands. The last two instructions

```
MOV AX, 4C00h   ;terminate the program
INT 21h
```

in code segment perform the task of program termination and returning control back to operating system. These two lines are examples of using DOS interrupt service to terminate a program.

The first step after editing and saving a program in a file, is to assemble it using the assembler. One of the most widely used 8086 assembler is Microsoft Assembler, popularly known as MASM. Assume that it is available in MASM folder on C drive of your machine; the following command will assemble the program

```
C:\MASM>masm add.asm
```

Once the **Enter** key is pressed, it will ask for the name of the object file with default name *add.obj* and other file names. Press **Enter** for all of them. If there are any syntax errors, they are listed with line number and message. If there are no errors, then it prepares the *.obj* file containing machine codes with unresolved references. In our case, the object file with name *add.obj* is prepared. The next step is to link the object file to get the executable file. The command

```
C:\MASM>link add.obj
```

will prompt you to enter the name of executable file with default name *add.exe*. It also asks for other file names. Press **Enter** for all of them to accept the default names. The *add.exe* file produced by the above command is a DOS executable file and can be executed from DOS prompt just like any other command. The following command executes the *add.exe*.

```
C:\MASM>add
```

Once the **Enter** key is pressed, the program executes and after completion the command prompt displays back. Surprisingly, the execution of addition program shows nothing on screen. The reason is that the program simply works with the internal registers whose contents are not displayed automatically on the screen. To see the contents of internal registers, we have to use the tool called debugger. The DOS provides debugger called "debug" for that purpose. It is not only used to see the results, but it also allows the user to execute program in step modes and thus facilitate to debug the program for logical errors. We will learn about it in detail in the next section.

We can also use the "masm" and "link" commands in slightly different manner to make our task simple. Assuming that our program is stored in *add.asm* file, the command

```
C:\MASM>masm add;
```

reads *add.asm* file and produces *add.obj* file without asking for any further names if there are no errors. The command

```
C:\MASM>link add;
```

reads *add.obj* file and produces *add.exe* file without asking any names.

In Program 3.1, we have mixed the data together with the code by writing a single segment for simplicity. It is better to separate the data and code taking advantage of segmentation for the ease of

management. Program 3.2 writes the listing of Program 3.1 using separate data and code segments. Assume that the code for Program 3.2 is stored in the file *add1.asm*.

Program 3.2

Rewrite Program 3.1 with separate data and code segments.

Solution

The program is

```
data    SEGMENT
        n1   DW 1234h      ;define n1 with value 1234h
        n2   DW 4321h      ;define n2 with value 4321h
        ans DW   ?         ;allocate space for answer
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data       ;initialize the data segment
        MOV DS, AX
        MOV AX, n1         ;load AX with n1
        MOV BX, n2         ;load BX with n2
        ADD AX, BX         ;add AX and BX
        MOV ans, AX        ;store result in ans

        MOV AX, 4C00h      ;terminate the program
        INT 21h

code    ENDS
        END start
```

Note that only CS register is automatically initialized to base address of the physical segment corresponding to the logical segment `code`. The offset address for the CS register is always provided by the Instruction Pointer (IP) register and the instructions are fetched from the current CS:IP address during execution starting from offset 0000h from the segment pointed by CS. Each time IP is incremented by the length of the instruction to point to next instruction in sequence. This is how the program is executed from first instruction to last instruction.

During the execution, instructions refer to data from the data segment. To get the correct values, the DS register must point to physical segment created to store our data values defined in logical segment `data`. However, the initialization of the DS register is not done automatically. This is to be performed by the user in beginning of the code segment before instructions refer any data from data segment. The following statements in code segment performs this task:

```
start:  MOV AX, data       ;initialize the data segment
        MOV DS, AX
```

In the first instruction, the name `data` is replaced by the segment base of the physical segment created to store the data. Then it is stored in DS register using AX register. The direct loading of immediate

value into segment register is not permitted and thus it is done using AX register as intermediate register. Once the initialization is completed by above instructions, the DS points to the correct physical segment and thereafter reference to any value in data segment made by instructions gets correct values.

The next four instructions after the initialization in code segment read the values `n1` and `n2` from data segment, add them and store the result into variable `ans`. The last two instructions perform the termination of the program.

Assemble Program 3.2 (stored in file *add1.asm*) using "`masm`" command, link it with "`link`" command to get the *add1.exe*. Executing *add1.exe* on command prompt executes it but does not show the results as the operations are done in internal register. In the next section, we will see that how we can use "`debug`" command to run it in step modes and see the results.

3.4 Debugging a Program

We know now how to convert an assembly language program to executable machine code using assembler and linker. We have seen in the previous section that executing *.exe* file does not show any results on the screen. This is where the debugging tool provided by DOS as a "`debug`" command comes to our help. It does not only show the results but provides very useful facilities like debugging a program to find the logical errors. The "`debug`" provides all these operations by its rich command set. To start the "`debug`", give the following command at the DOS prompt:

```
C:\MASM>debug
-
```

Pressing **Enter** will get the debug prompt "-". The debug is now ready to accept its valid command from its command set. Type **"?"** at the "`debug`" prompt to get the list of its command set. We will use some of them in this section to understand the working of programs in details using executable file *add1.exe* for Program 3.2.

Unassemble the Machine Codes

To load the *add1.exe* using debug, issue the command "`debug add1.exe`" at the DOS prompt as shown in Fig. 2. Once the executable file is loaded, we can perform various operations on that by issuing `debug` commands at the debug prompt "-". One of the most useful commands is "`u`" (unassemble) which unassembles the machine code to assembly code and displays it. If it is issued without any address range, then it starts from the current offset address; otherwise it gives the code in given offset range.

The first command issued to debug in Fig. 2 is "`u 0 14`" which unassembles the instructions from the offset address 0000h to 0014h covering entire body of code segment of Program 3.2. Each line shows the physical address in `segment:offset` form, then machine codes and the assembly code. It is seen in the figure that the code segment starts from the physical address 0B59:0000h. This means that the segment base for code segment is 0B59h and is loaded in CS register automatically by OS.

Look to the Data Segment

Observe from the code in Fig. 2 that all the names are replaced by their address values, either segment or offset. The segment name `data` in first instruction is replaced by segment base 0B58h indicating that the data segment is started at the physical address 0B58:0000. The image of the data segment is

```
C:\MASM>debug add1.exe
-u 0 14
0B59:0000 B8580B          MOV     AX,0B58
0B59:0003 8ED8            MOV     DS,AX
0B59:0005 A10000          MOV     AX,[0000]
0B59:0008 8B1E0200        MOV     BX,[0002]
0B59:000C 03C3            ADD     AX,BX
0B59:000E A30400          MOV     [0004],AX
0B59:0011 B8004C          MOV     AX,4C00
0B59:0014 CD21            INT     21
-
-e 0B58:0000
0B58:0000  34.     12.     21.     43.     00.     00.
-
```

Figure 2 Physical segments in memory for Program 3.2.

shown in the figure at the bottom by giving the "e" (enter) command with the starting address at the debug prompt. The enter command lists the subsequent values by pressing spacebar until **Enter** key is pressed. We have listed only first six bytes that are values of three variables n1, n2 and ans. Observe that the offset of n1 is 0000h, offset of n2 is 0002h and offset of ans is 0004h in the segment with base 0B58h. The references to these variables are replaced by their respective offset addresses using direct addressing mode as shown in the figure. For example, the instruction

```
MOV AX, n1
```

is built as

```
MOV AX, [0000h]
```

as the offset address of n1 is 0000h.

Comparing the code listed in Program 3.2 with its counterpart in Fig. 2 gives an idea of how the conversion process takes place and the layout of the physical segments corresponding to the logical segments defined in the program. The important fact we can observe here is that all the assembler directives are removed as their role is over once the conversion is completed. All the names including segment names and variable names are replaced by numbers, that is, addresses (segment base or offsets) as machine language consists of only numbers.

Displaying and Changing Register Contents

Using the "r" (register) command in the debug command set, we can display the contents of all the registers. Figure 3 show the use of the "r" command after loading the *add1.exe*.

As shown in the figure, the first "r" command shows the contents of all the registers including nine flags in first two lines and the current instruction pointed by IP register in third line. It is the first instruction in Program 3.2 as the IP = 0000h. This is the instruction to be executed next. The second command is "r AX" which displays the contents of AX register and then after displaying colon (:) waits for the user to enter new contents for that register. We have entered the value 1234 to load into the AX register. This is visible after giving the third "r" command. The content of AX is changed to 1234h. The forth command "r F" lists the current status of binary flags and then displays "-" to enter the modified status for any of them. We have changed the status of overflow flag from reset to set which is visible after giving the fifth "r" command. The notations used by the debug for set and reset conditions of the flags are shown in Table 1.

```
C:\MASM>debug add1.exe
-r
AX=0000  BX=0000  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B48  ES=0B48  SS=0B58  CS=0B59  IP=0000   NV UP EI PL NZ NA PO NC
0B59:0000 B8580B        MOV     AX,0B58
-r AX
AX 0000
:1234
-r
AX=1234  BX=0000  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B48  ES=0B48  SS=0B58  CS=0B59  IP=0000   NV UP EI PL NZ NA PO NC
0B59:0000 B8580B        MOV     AX,0B58
-r F
NV UP EI PL NZ NA PO NC  -OV
-r
AX=1234  BX=0000  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B48  ES=0B48  SS=0B58  CS=0B59  IP=0000   OV UP EI PL NZ NA PO NC
0B59:0000 B8580B        MOV     AX,0B58
-
```

Figure 3 Displaying and modifying register contents.

Table 1 Flag notations used by debug

Flag	*Set (=1)*	*Reset (=0)*
CF	CY	NC
PF	PE	PO
AF	AC	NA
ZF	ZR	NZ
SF	NG	PL
OF	OV	NV
DF	DN	UP
IF	EI	DI

Running Program in Step Mode

Debug provides very useful facility using "t" (trace) command. It allows us to run a program instruction at a time and display the contents of registers and memory for visual inspection. The 8086 contains the trap flag which when set to 1, the processor stops after execution of an instruction. The debug internally sets the trap flag to 1 and then executes an instruction when the "t" command is issued. This is useful to debug the program for any logical errors by running it in step-wise manner also called single-step mode.

Figure 4 shows the complete trace of Program 3.2 (*add1.exe*) using step mode. The first "r" command displays the register contests and the instruction pointed by IP. Observe that CS contains 0B59h and IP contains 0000h, together pointing to physical address 0B59:0000h where the first instruction is stored. Carefully observe that DS register is not yet initialized to correct address as it is to be done by instructions which are not yet executed. The next command "t" executes the first instruction and displays the register contents after execution. You can observe the change in contents of AX register as well as IP register. AX is loaded with 0B58h and IP is incremented to point to next instruction which is also displayed in output. After executing first two instructions, the DS is initialized to proper segment

base. Each "t" command executes the next instruction and shows the register contents and the next instruction to be executed. See that if the instruction to be executed refers the data from memory, it is also shown in right-hand side after the instruction. When IP = 0014h, the instruction to be executed is INT 21h which is the Interrupt Service Routine (ISR) for interrupt service 21h. If you press "t" then debug will start executing the ISR in step mode. The ISR is already a well-tested service and we need to use it rather than debug it. Hence, at that time we need to use "p" (procedure) command which executes whole routine at a time rather than "t" command. It is end of the program and the message

```
C:\MASM>debug add1.exe
-r
AX=0000  BX=0000  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B48  ES=0B48  SS=0B58  CS=0B59  IP=0000   NV UP EI PL NZ NA PO NC
0B59:0000 B8580B        MOV     AX,0B58
-t

AX=0B58  BX=0000  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B48  ES=0B48  SS=0B58  CS=0B59  IP=0003   NV UP EI PL NZ NA PO NC
0B59:0003 8ED8          MOV     DS,AX
-t

AX=0B58  BX=0000  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B58  ES=0B48  SS=0B58  CS=0B59  IP=0005   NV UP EI PL NZ NA PO NC
0B59:0005 A10000        MOV     AX,[0000]                    DS:0000=1234
-t

AX=1234  BX=0000  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B58  ES=0B48  SS=0B58  CS=0B59  IP=0008   NV UP EI PL NZ NA PO NC
0B59:0008 8B1E0200      MOV     BX,[0002]                    DS:0002=4321
-t


AX=1234  BX=4321  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B58  ES=0B48  SS=0B58  CS=0B59  IP=000C   NV UP EI PL NZ NA PO NC
0B59:000C 03C3          ADD     AX,BX
-t

AX=5555  BX=4321  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B58  ES=0B48  SS=0B58  CS=0B59  IP=000E   NV UP EI PL NZ NA PE NC
0B59:000E A30400        MOV     [0004],AX                    DS:0004=0000
-t

AX=5555  BX=4321  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B58  ES=0B48  SS=0B58  CS=0B59  IP=0011   NV UP EI PL NZ NA PE NC
0B59:0011 B8004C        MOV     AX,4C00
-t

AX=4C00  BX=4321  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B58  ES=0B48  SS=0B58  CS=0B59  IP=0014   NV UP EI PL NZ NA PE NC
0B59:0014 CD21          INT     21
-p

Program terminated normally
-e ds:0000
0B58:0000  34.     12.     21.     43.     55.     55.
-
```

Figure 4 Complete trace of Program 3.2 using debug.

```
-e ds:0000
0B58:0000  34.35   12.A2   21.55   43.FF   55.00   55.00
-r ip
IP 0014
:0000

-r
AX=4C00  BX=4321  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B58  ES=0B48  SS=0B58  CS=0B59  IP=0000   NV UP EI PL NZ NA PE NC
0B59:0000 B8580B        MOV     AX,0B58
-
```

Figure 5 Running program again with different values.

"`Program terminated normally`" is displayed. Finally, using "`e`" command, the contents of data segment are displayed showing that the result 5555h is stored at offset 0004h and 0005h, the storage space of the variable `ans`.

It is possible to re-run the program using debug with different values of `n1` and `n2` without changing the source code and reassembling it. The "`e`" command also allows you to change the contents of memory. While it displays the current value, it prompts us to also change the value. Before pressing the spacebar to move to the next value, enter the new value. Let us assume that we want to run *add1.exe* again after completing the execution shown in Fig. 4. First we will enter the new values for `n1` and `n2` as A234h and FF55h, respectively, using "`e`" command as shown in Fig. 5. The IP is pointing to the end of the program which needs to be initialized to first instruction by changing it to 0000h using "`r`" command. It now points to the first instruction as shown in Fig. 5. Now simply use "`t`" command to execute the program with new values in the same manner and finally, verify the result using "`e`" command.

Go Command

While debugging a large and complex program, we need to run it multiple times which is time consuming. Specifically, programs using loops with a large number of iteration cause this issue. Debug provides the solution to this problem using "`g`" (go) command. The "`g`" command allows you to run block of instructions from current offset to the given offset known as *breakpoint*. In a large program, we can run up to breakpoint using single "`g`" command and then we use trace mode to debug the rest of the part. Each time part of program is corrected, we can further move the breakpoint. This will save the time of debugging large programs. Figure 6 shows the use of `go` command. It runs the program up to ADD instruction using "`g 000C`" command where 000Ch is the offset of the ADD instructions. It runs all the instructions from offset 0000h up to offset 000Ch. It then shows the register contents and next instruction to be executed.

Quit the Debug

Finally when the use of debug is over, the "`q`" (quit) command is used to quit from the debug to go back to the DOS prompt.

We have used only those debug commands which are necessary to debug a program. However, debug provides many other useful commands. The interested readers may refer the online material or try them with debug to learn them.

```
C:\MASM>debug add1.exe
-u 0 14
0B59:0000 B8580B              MOV     AX,0B58
0B59:0003 8ED8                MOV     DS,AX
0B59:0005 A10000              MOV     AX,[0000]
0B59:0008 8B1E0200            MOV     BX,[0002]
0B59:000C 03C3                ADD     AX,BX
0B59:000E A30400              MOV     [0004],AX
0B59:0011 B8004C              MOV     AX,4C00
0B59:0014 CD21                INT     21
-
-g 000C
AX=1234  BX=4321  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B58  ES=0B48  SS=0B58  CS=0B59  IP=000C   NV UP EI PL NZ NA PO NC
0B59:000C 03C3          ADD     AX,BX
-
```

Figure 6 Use of the go command.

Check Points

We now know that

- the 8086 assembly language program consists of one or more logical segments.
- the assembly language program is written using any text editor with filename having *.asm* extension.
- *.asm* file is assembled using assembler to get the *.obj* file which is linked using linker to get the *.exe* (executable) file.
- executable file is run on DOS like any other command.
- DOS provides debugger called "debug" which is very useful to run the programs in step modes and debug to find the logical errors.
- The `unassemble`, `enter`, `register`, `trace`, `go` and `quit` are commonly used commands from debug command set.

3.5 Using DOS Service for Character and String I/O

The first Personal Computer – designed by IBM known as IBM PC – used Intel's 8088 as processor and Microsoft Disk Operating System (MS DOS) as the operating system. As the underlying processor is the 8088, the DOS is designed around the 8088 interrupts. We know that the 8086 and 8088 are software compatible which means that program written for one works on other also. We can access the DOS services using INT instruction in our program. Each interrupt service is given an 8-bit number. To access particular service, we have to use its 8-bit number with INT instruction which executes the ISR for that service. For example,

```
INT 21h
```

invokes the ISR for service 21h. Each interrupt service provides the number of functions. Each function is also given an 8-bit number which is to be specified in the AH register. The service 21h is the major DOS service providing large number of functions. Before calling a function of the service, the

required input parameters are to be given in specified registers. If the function called returns the output, it comes in specified registers which can be looked after invoking service through INT instruction. The format for calling a function from particular service is as follows:

```
MOV AH, fun_num
;give the input parameters
;in specified registers
INT ser_num
;results are in specified registers
```

The fun_num and ser_num are 8-bit function number and service number, respectively. The input and output registers for each function in a service are predefined and available in DOS manuals.

We have already used such service in our previous programming examples to terminate the program as

```
MOV AX, 4C00h
INT 21h
```

which is same as

```
MOV AH, 4Ch        ;function no = 4Ch
MOV AL, 00h        ;input parameter AL = 0
INT 21h            ;invoke the service
```

This specifies the function number 4Ch which "terminates a program", AL = 0 specifies the input parameter and finally INT 21h invokes the desired function from service 21h and terminates a program.

We will see how to use I/O functions to read and write characters and strings from service 21h using programs. We will learn more about DOS interrupt services in Chapter 7 while learning about the 8086 interrupts.

Character I/O

The DOS service 21h provides the functions for reading a character from keyboard and writing a character to display. The details of the function to read a character are as follows.

Function	: Read a character with echo
Function number	: AH = 1
Input	: None
Output	: AL = ASCII of character pressed

The character given as input from the keyboard is echoed on the screen. It does not need any input and the ASCII value of the character entered is returned in the AL register. It can be used to get a character as follows:

```
MOV AH, 1      ;function number
INT 21h        ;call the service
```

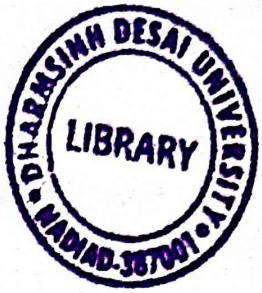
After loading AH with function number = 1 and calling the service by executing INT 21h instruction, it stops for user to enter a character. Once the character is pressed it is echoed on the screen and its ASCII value is stored in the AL register which can be used by subsequent instructions in the program as per need. For example, pressing a key 'a' echoes character 'a' on the screen and puts its ASCII value 61h in the AL register.

Function 2 of the service 21h is used to display a character on the screen whose ASCII value is given as input in the DL register. The details of function to display a character are as follows.

Function	: Display a character
Function number	: AH = 2
Input	: DL = ASCII of character to display
Output	: None

Following code will display, capital alphabet 'A' on the screen as its ASCII is 41h.

```
MOV AH, 2              ;function number
MOV DL, 41h            ;load DL with 'A'
INT 21h                ;call the service
```



Program 3.3

Write a program to convert a given alphabet from lowercase to uppercase.

Solution

The program is

```
code    SEGMENT
        ASSUME cs:code
start:  MOV AH, 1          ;read a character
        INT 21h
        SUB AL, 20h        ;convert lower to upper
        MOV AH, 2          ;print a character
        MOV DL, AL
        INT 21h
        MOV AX, 4C00h      ;terminate the program
        INT 21h
code    ENDS
        END start
```

The ASCII of lowercase letter starts from 61h (97) and that of uppercase from 41h (65). Subtracting 20h (32) from the ASCII of lowercase letter converts it to uppercase. Assume that the above code is stored in the file *char_con.asm;* after assembling and linking, the *char_con.exe* is created. Running it on DOS prompt shows the result as

```
C:\MASM>char_con
aA
C:\MASM>
```

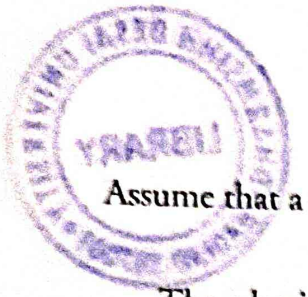
Once a program starts running, it waits for the user to enter a character. We have entered 'a' which is echoed on screen by read function and then it is converted to uppercase by subtract instruction and finally display function display it as 'A'.

Display a String

A string is a sequence of characters stored as ASCII values. The function 9 in DOS service 21h provides the facility to display strings on the screen. We can use it to display messages while running a program to make it interactive. Function 9 needs the starting address of the string to be displayed in DS:DX

(segment:offset) and string is to be ended by '$' character. It prints characters starting from address DS:DX until the character is '$'. The details of the function to display a string are as follows:

Function	: Display a string
Function number	: AH = 9
Input	: DS = segment address of string DX = offset address of string
Output	: None



Assume that a string mes containing "Hello, World$" is defined as follows in a data segment.

```
mes     DB    "Hello, World$"
```

The physical address of the string is defined by segment and offset address of string variable mes which we can get using the SEG and OFFSET directives. The following code displays the above string:

```
MOV AH, 9                ;function number
MOV BX, SEG mes          ;get segment address
MOV DS, BX
MOV DX, OFFSET mes       ;get offset address
INT 21h
```

Observe that the segment address is copied into DS register using BX register because the immediate transfer to segment registers is not permitted . Keep in mind that all the register values including function number are to be preserved until service is called by INT 21h instruction. For example, by mistake if you use AX to transfer the segment address into DS, the function number loaded in AH gets overwritten and the service gets improper function number.

Program 3.4

Write a program to display message "Hello, World".

Solution

The program is

```
data    SEGMENT
        mes         DB    "Hello, World$"
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data              ;initialize the data segment
        MOV DS, AX
        MOV AH, 9                 ;function number
        MOV BX, SEG mes           ;get segment address
        MOV DS, BX
        MOV DX, OFFSET mes        ;get offset address
        INT 21h
        MOV AX, 4C00h             ;terminate the program
```

```
        INT 21h
code    ENDS
        END start
```

Assume that the Program 3.4 is stored in *hello.asm*, the assembling and linking will produce the file *hello.exe*. The execution of it shows the output as follows:

```
C:\MASM>hello
Hello, World
C:\MASM>
```

Figure 7 shows the code and data segment layout of Program 3.4. Observe that directives SEG and OFFSET are replaced by the values. The layout of data segment at address 0B5E:0000h shows the storage of "Hello, World$" as ASCII values starting with 65h (ASCII of H) and ending with 24h (ASCII of $).

```
C:\MASM>debug hello.exe
-u 0 14
0B5F:0000 B85E0B              MOV     AX,0B5E
0B5F:0003 8ED8                MOV     DS,AX
0B5F:0005 B409                MOV     AH,09
0B5F:0007 BB5E0B              MOV     BX,0B5E
0B5F:000A 8EDB                MOV     DS,BX
0B5F:000C BA0000              MOV     DX,0000
0B5F:000F CD21                INT     21
0B5F:0011 B8004C              MOV     AX,4C00
0B5F:0014 CD21                INT     21
-
-e 0B5E:0000
0B5E:0000  48.    65.    6C.    6C.    6F.    2C.    20.    57.
0B5E:0008  6F.    72.    6C.    64.    24.
-
```

Figure 7 Code and data layout of Program 3.4.

If you examine the code shown in Fig. 7, it shows that first two instructions used for initialization as well as the fourth and fifth instructions used to get the segment address of mes transfer the same segment value 0B5Eh in DS as both place we refer to same segment. In this case, we can remove the instructions to get the segment address of mes to avoid duplication. We can simply write the code to display string as

```
MOV AH, 9               ;function number
MOV DX, OFFSET mes      ;get offset address
INT 21h
```

This will save the execution time. However, you must be careful that this is not always the case. If you are referring to different segment, it is necessary to write it exclusively, otherwise problems may arise.

Function 9 of service 21h not only displays the printable ASCII characters, it also processes the non-printable ASCII characters by performing the function defined by ASCII character. For example, cursor movement functions like carriage return and line feed are defined by ACSII values 0Dh and 0Ah, respectively. Both together define a new line. For example, the following when printed prints "Hello" and "World" in separate lines:

```
mes     DB      "Hello", 0Dh, 0Ah, "World$"
```

First the word "Hello" is printed, then the 0Dh and 0Ah characters cause the cursor to move to the beginning of the next line and finally the word "World" is printed in the second line.

Read a String

Function 10 (0ah) of service 21h is used to read a string from the keyboard. To store the string read by this function, the DOS needs storage, that is, buffer for that in specific format. Figure 8 shows the format for the string buffer for the function 10.

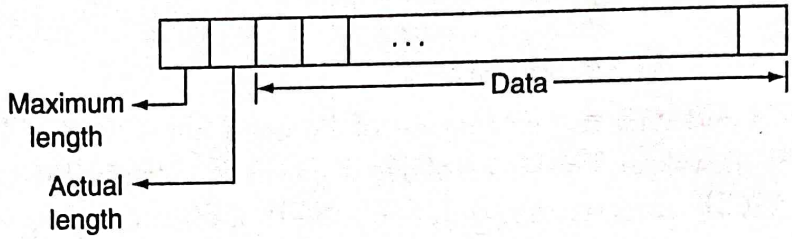


Figure 8 Format of string buffer.

It is divided into three fields. The first 8-bit field decides the maximum length of the input string. The second 8-bit field stores the actual length of the string once it is entered. The third field is data field which really stores the string as a sequence of ASCII values. The largest value of maximum length field can be FFh (255) as it is 8-bit which means that the data field can contain largest string having length of 255 characters including the **Enter** key pressed to end the string. The **Enter** key is stored as 0Dh by the DOS. We can define the string buffer with maximum length as follows:

```
str_buf  DB  255, 256 dup(0)
```

This defines a string buffer with name `str_buf` having value 255 for maximum length and initializes the actual length and the data area of 255 bytes with 0 as shown in Fig. 9(a). Assume that the user enters a string "Hello"; when service to read a string is called, the contents of `str_buf` will become as shown in Fig. 9(b). Observe that the string length is automatically counted by the function and entered into the actual length field which we can later use in the program by reading it. Note that the **Enter**

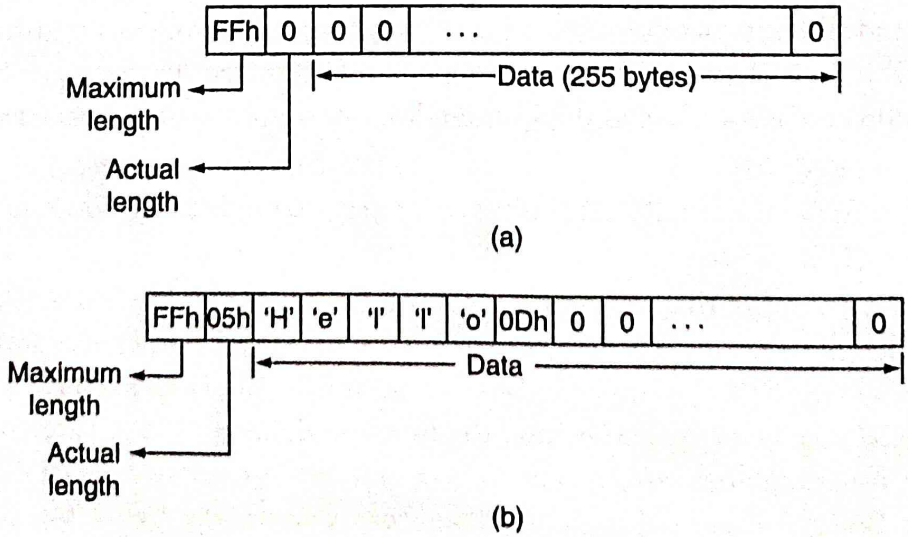


Figure 9 Contents of string buffer (a) before and (b) after reading a string.

key (0Dh) is not counted in the length. The ASCII values of string characters starts from `str_buf+2` position in string buffer as first two bytes are occupied by maximum length and actual length.

The function 10 needs the address of string buffer defined to store an input string in the DS:DX register in same manner as function to display a string. The details of function to read a string are as follows:

Function : Read a string
Function number : AH = 10
Input : DS = segment address of string buffer
DX = offset address of string buffer
Output : String is stored in string buffer

We can write the code to read the string as follows.

```
MOV AH, 10                          ;function number
MOV BX, SEG str_buf                 ;get segment address
MOV DS, BX
MOV DX, OFFSET str_buf              ;get offset address
INT 21h
```

This is same as displaying a string except the function number. Once the above code is executed, the program stops to read a string from keyboard. It accepts the characters (maximum 255) until **Enter** key is pressed, stores them in data part of the buffer and also updates the actual length field with length of the string entered.

Program 3.5

Write a program to read a string from the keyboard and display it on the screen.

Solution

The program is

```
data    SEGMENT
        mes1      DB   "Enter a string : $"
        str_buf   DB   255,256 dup(0)
        nl        DB   0Dh, 0Ah, '$'
        mes2      DB   "String entered is : $"
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data      ;initialize the data segment
        MOV DS, AX
        MOV AH, 9         ;print message
        MOV DX, OFFSET mes1
        INT 21h
        MOV AH, 10        ;read a string
        MOV DX, OFFSET str_buf
        INT 21h
```

```
        MOV SI, OFFSET str_buf+1 ;replace end of string
        MOV CX, 0                ;with '$'
        MOV CL, BYTE PTR [SI]
        INC SI
        ADD SI, CX
        MOV BYTE PTR [SI], '$'
        MOV AH, 9                ;print new line
        MOV DX, OFFSET nl
        INT 21h
        MOV AH, 9                ;print message
        MOV DX, OFFSET mes2.
        INT 21h
        MOV AH, 9                ;print a string
        MOV DX, OFFSET str_buf+2
        INT 21h
        MOV AX, 4C00h            ;terminate the program
        INT 21h
code    ENDS
        END start
```

As seen in Program 3.5, first the string is read from the user. We know that the end of the string is marked as 0Dh by the function 10. In order to print it using function 9, we have to change it to '$'. This is performed after reading string through the block of six instructions. The first – the offset address of actual length field – is loaded into SI, the length is read from that offset into CL register, the SI is moved to point to first character in data field and then adding length in CX to SI moves the SI to point to the position where end of the string is stored. The last instruction replaces it by the '$' character. Another important thing is that while finally printing an entered string using function 9, the offset of `str_buf+2` is moved to DX as the actual string characters start from that position in `str_buf`.

Check Points

We now know that

- the services of DOS are accessible in the 8086 programs using INT instruction by specifying the function number and service number.
- the service 21h is the most important DOS service that provides number of useful functions.
- function 1 and 2 of the service 21h are used for character reading and writing, respectively.
- function 9 of service 21h is used to display a string whereas function 10 is used to read string.

When executed, Program 3.5 produces the output as follows:

```
Enter a string : Hello, World
Entered string is : Hello, World
```

3.6 Data Transfer Instructions

The data transfer instructions are used to move the operands between registers, registers and memory, and registers and I/O ports. The data transfer instructions copies the operands from one place to other, but do not manipulate them. Hence, they do not affect the flags, that is, the contents of the flag registers remain unchanged after performing any of the data transfer instruction. We will learn the general and frequently used data transfer instructions in this section whereas other data transfer instructions will be covered in subsequent chapters during the discussion of relevant topics.

MOV Instruction

It is the most frequently used instruction of the 8086 instruction set. We have already used MOV instruction while discussing the instruction templates and addressing modes in previous chapter. It is also used in programs we have written in this chapter. The syntax for the MOV instruction is as follows:

```
MOV destination, source
(destination) ← (source)
```

The ( ) brackets represent "the contents of"; if the operand is a register, then meaning is contents of that register and if the operand is memory location, then meaning is contents of that memory location. The contents of the `source` operand are copied to the `destination` operand. The contents of `source` operand remain unchanged. The operands are either byte or word operands and *both must be of same type in an instruction.* The `source` can be immediate value, register or memory location. The `destination` can be register or memory location. The `destination` cannot be immediate value as it does not represent storage. The possible combinations for the `source` and `destination` operands are as follows.

Destination	*Source*
Register	Register
Register	Memory
Register	Immediate value
Memory	Register
Memory	Memory
Memory	Immediate value

Observe that last two combinations represent memory storage for both *source* and *destination* operands. Remember that immediate value is part of instruction and stored in memory along with opcode of the instruction. The 8086 instructions do not permit memory to memory transfers or in general both the operands cannot be in memory. This makes the last two combinations invalid for use. When we refer to the register operand, mostly we refer to general-purpose registers, pointer registers and index registers. Whenever a register operand refers to any segment register, some restrictions are to be observed. For example, the immediate to register transfer is not permitted for segment registers although it is a valid combination. These concepts remain same for almost all the 8086 instructions using two operands in a format like MOV. Hence, each time we will not discuss the same concepts, exceptions, if any, will be mentioned.

XCHG Instruction

It is known as exchange instruction and used to exchange the contents of operands. The syntax for the XCHG instruction is as follows.

```
XCHG destination, source
(destination) ↔ (source)
```

The contents of *source* and *destination* operands are swapped (i.e., exchanged) with each other. The *source* can be register or memory location. The *destination* can be register or memory location. The *source* and *destination* both cannot represent the memory locations. Operands must be of same type either bytes or words. Following are some examples:

```
XCHG AX, BX                ;exchange AX and BX
XCHG CL, BYTE PTR [SI]     ;exchange CL and byte at [SI]
XCHG [SI+2], AX            ;exchange word at [SI+2] and AX
XCHG DX, n1                ;exchange DX and word defined by n1
```

Program 3.6

Write a program to exchange the contents of two words stored in memory.

Solution

The program is

```
data    SEGMENT
            n1      DW      1234h
            n2      DW      4321h
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data        ;initialize the data segment
        MOV DS, AX

        MOV AX, n1          ;get n1 to AX
        XCHG AX, n2         ;exchange AX and n2
        MOV n1, AX          ;store AX(n2) in n1

        MOV AX, 4C00h       ;terminate the program
        INT 21h

code    ENDS
        END start
```

XLAT Instruction

It is known as translate instruction. It is mainly used to translate the codes from one system to another using lookup table. The syntax for the XLAT instruction is as follows:

```
XLAT
(AL) ← (BX + AL)
```

The byte stored in AL register is replaced by the byte stored at the location BX + AL in the lookup table where BX is the base of the lookup table and contains offset address of the lookup table stored in data segment. Program 3.7 clears the concept by converting lowercase string to uppercase using lookup table and XLAT instruction.

Program 3.7

Write a program to convert lowercase string to uppercase using XLAT instruction. Input string should not contain characters other than lowercase alphabets including spaces.

Solution

The program is

```
data    SEGMENT
        mes1    DB  "Enter a lower case string : $"
        str_buf DB  255,256 dup(0)
        nl      DB  0Dh, 0Ah, '$'
        mes2    DB  "Uppercase string is : $"
        table   DB  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data      ;initialize the data segment
        MOV DS, AX

        MOV AH, 9   ;print message
        MOV DX, OFFSET mes1
        INT 21h

        MOV AH, 10  ;read a string
        MOV DX, OFFSET str_buf
        INT 21h
        ;store the base of table to BX
        MOV BX, offset table
        ;get the length of string into CL
        MOV SI, OFFSET str_buf+1
        MOV CX, 0
        MOV CL, BYTE PTR [SI]
next:   INC SI                    ;point next character
        MOV AL, BYTE PTR [SI]     ;get the character
        SUB AL, 'a'               ;convert to index
        XLAT                      ;translate to uppercase
        MOV BYTE PTR [SI], AL     ;store back
        LOOP next                 ;repeat if not end
```

```
        INC SI                      ;go to next position
        MOV BYTE PTR [SI], '$'      ;end the string with '$'

        MOV AH, 9                   ;print new line
        MOV DX, OFFSET nl
        INT 21h

        MOV AH, 9                   ;print message
        MOV DX, OFFSET mes2
        INT 21h

        MOV AH, 9                   ;print a string
        MOV DX, OFFSET str_buf+2
        INT 21h

        MOV AX, 4C00h               ;terminate the program
        INT 21h

code    ENDS
        END start
```

As seen in Program 3.7, after reading a string, the code for conversion using lookup table and XLAT instruction is written and then the converted string is printed. For conversion, first the offset of lookup table defined as `table` is get into the BX register. The length of the string is stored in the CX register for controlling the number of iterations using LOOP instruction. LOOP instruction repeats the code from the instruction labelled with label `next` up to LOOP instruction CX number of times. Each time it reduces the CX by 1 and if CX ≠ 0, it jumps to `next` and repeats the operation. When CX = 0, the loop is terminated. Each time in a loop the current character pointed by SI is moved into the AL, the ASCII of 'a' is subtracted from it to convert it into the index (index of 'a' is 0, 'b' is 1, ..., 'z' is 25) and then it is converted to uppercase letter using XLAT instruction. For example, if character is 'd', then the index is 3 and the character at BX+3 (fourth character in `table`) is 'D'. The converted character is stored back in the original string at SI. After converting whole string in this manner, it is terminated with '$' so that we can print it properly. The execution of program produces the results as

```
Enter a lowercase string : mansukh
Uppercase string is : MANSUKH
```

Run this program in the debug using `trace` command to get the idea about how loops are executed. When the LOOP instruction executes and if CX ≠ 0, the control is transferred back to the start of loop. You can observe this by looking at the value of IP register and the next instruction to be executed. When CX = 0, the control goes to the next instruction after LOOP instruction.

LEA, LDS and LES Instructions

These are address transfer instructions and are used to load the registers with the addresses. They are explained as follows.

LEA Instruction

LEA is known as "Load Effective Address" instruction and is used to transfer the effective address, that is, offset of memory operand, into specified register. The syntax of LEA instruction is as follows:

```
LEA destination, source
(destination) ← effective address specified by source
```

The *source* is always memory location and the *destination* can be any of the 16-bit general-purpose, pointer or index registers. Following are some examples.

```
LEA BX, [SI]        ;load effective address in SI to BX
LEA DI, [BX+SI]     ;load effective address computed by
                    ;adding BX and SI into DI
LEA SI, n           ;load effective address of n into SI
```

In previous programs, we have used instructions like

```
MOV DX, OFFSET mes
```

which can be written using LEA as

```
LEA DX, mes
```

LDS Instruction

The LDS instruction is known as "Load register and DS register". It loads both the register specified and the DS register from the memory location specified. The syntax of the LDS instruction is as follows:

```
LDS destination, source
(destination) ← (source)
(DS) ← (source+2)
```

The *source* is always memory location whereas the *destination* can be any of the 16-bit general-purpose, pointer or index registers. It copies the word from the memory location specified by the *source* to the register specified as *destination* and the consecutive word at the memory location specified by *source* plus 2 in the DS register. This instruction is generally used to initialize the SI and DS registers with the offset and segment address of the source string in case of string instructions. Assume that BX contains the value 4400h, then the instruction

```
LDS SI, [BX]
```

copies the word made by the bytes at offset address 4400h and 4401h to the SI register and the word made by the bytes at offset address 4402h and 4403h to the DS register.

LES Instruction

The LES instruction is known as "Load register and ES register". It loads both the register specified and the ES register from the memory location specified. The syntax of the LES instruction is as follows:

```
LES destination, source
(destination) ← (source)
(ES) ← (source+2)
```

The *source* is always memory location whereas the *destination* can be any of the 16-bit general-purpose, pointer or index registers. It copies the word from the memory location specified by the *source* to the register specified as *destination* and the consecutive word at the memory location specified by *source* plus 2 in the ES register. This instruction is generally used to initialize the DI and ES registers with the offset and segment address of the destination string in case of the string instructions. Assume that BX contains the value 4400h, then the instruction

```
LES DI, [BX]
```

copies the word made by the bytes at offset address 4400h and 4401h to the DI register and the word made by the bytes at offset address 4402h and 4403h to the ES register.

LAHF and SAHF Instructions

We know that the flag register of the 8086 processor contains nine flags including six conditional flags and three control flags. The lower byte of the flag register includes five conditional flags including CF, PF, AF, ZF and SF. The 8085 processor contains these five flags in the same format in its flag register. Hence, we can say that the lower bye of the 8086 flag register is exactly compatible to the 8085 flag register. The LAHF and SAHF instructions are useful to simulate the 8085 on the 8086-based system.

The LAHF instruction is known as "Load AH with lower byte of flag register". It copies the lower byte of the 8086 flag register to the AH register. The syntax of the LAHF instruction is as follows:

```
LAHF
(AH) ← lower byte of flag register
```

The SAHF instruction performs reverse operation and is known as "Store AH to lower byte of flag register". It copies the contents of AH register into the lower byte of the 8086 flag register. The syntax of the SAHF instruction is as follows:

```
SAHF
lower byte of flag register ← (AH)
```

The rest of the instructions in the data transfer group include the stack-related instructions such as PUSH, POP, PUSHF and POPF as well as I/O instructions IN and OUT to deal with I/O ports. We will discuss them in later chapters while discussing the related topics.

Check Points

We now know that

- the data transfer instructions are used to transfer data between registers, registers and memory and register and I/O ports.
- they do not affect the flags as they do not manipulate operands.
- the MOV is the most frequently used instruction.
- the XCHG is used to swap the operands while XLAT is used for translation.
- LEA, LDS and LES are the address transfer instructions.
- LAHF and SAHF are used to load and store the 8085 compatible flags to AH and from AH, respectively.

3.7 Arithmetic Instructions

The arithmetic operations are most important operations in any programming. The 8086 processor provides a rich set of arithmetic instructions to perform add, subtract, multiply and divide operations on byte and word operands. It not only supports these operations on binary numbers, but also provides the instructions to handle the operands in the BCD form – both packed and unpacked. The arithmetic operations manipulate the operand values and hence they affect the conditional flags. This means that the contents of the flag register are updated based on the result of the operation. We will learn the simple arithmetic instructions handling binary operands in this section, while rest of the instructions dealing with BCD formats will be discussed in the next chapter. The arithmetic instructions are explained as follows.

Addition Instructions

The addition instructions cover the ADD (add), ADC (add with carry) and INC (increment) instructions. Let us learn them in detail with examples.

ADD Instruction

The ADD instruction is used to perform the arithmetic addition of the operands. The syntax for the ADD instruction is as follows:

```
ADD destination, source
(destination) ← (destination) + (source)
```

The contents of *source* and *destination* are added and the result is stored in *destination*. The *source* can be immediate value, register or memory location. The *destination* can be register or memory location. Both *source* and *destination* cannot be in memory. All the six conditional flags are updated based on the result stored in *destination*. Following are some examples:

```
ADD AX, BX                ;if AX = 5623h and BX = A236h
                          ;result in AX = F859h
                          ;OF=0, CF=0, PF=1, AF=0, ZF=0, SF=1
ADD AL, 35h               ;(AL) = (AL)+35h
ADD SI, WORD PTR [BX]     ;add SI with word at offset BX
ADD BYTE PTR [SI], 2      ;add 2 to the byte at offset SI
ADD AL, n                 ;add byte variable n to AL
```

Program 3.8

Write a program to add five words.

Solution

The program is

```
data    SEGMENT
        block DW  1234h, 5634h, 00D2h, 23A1h, 0AA45h
              DW  ?
```

```
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data            ;initialize the data segment
        MOV DS, AX
        MOV CX, 5               ;load the count
        MOV AX, 0               ;intialize sum
        LEA BX, block           ;point to first number
next:   ADD AX, WORD PTR [BX]   ;add next number
        ADD BX, 2               ;move to next number
        LOOP next               ;go to next
        MOV WORD PTR [BX], AX   ;store the result
        MOV AX, 4C00h           ;terminate the program
        INT 21h
code    ENDS
        END start
```

As seen in the program, CX is initialized to 5, AX with 0 and BX to point to first word. Each time in the loop, the word pointed by BX is added to AX and BX in incremented by 2 to point to next word. After completing the loop, the result in AX is stored to the next word position in memory.

ADC Instruction

The ADC is known as "add with carry". The syntax for the ADD instruction is as follows.

ADD *destination, source*
(*destination*) ← (*destination*) + (*source*) + (CF)

It adds the contents of the *source* and *destination* plus carry flag CF and stores the result into the *destination*. The rest of the things are same as the ADD instruction. Following are some examples:

```
ADC AX, BX          ;if AX = 5623h, BX = A236h and CF = 1
                    ;result in AX = F85Ah
                    ;OF=0, CF=0, PF=1, AF=0, ZF=0, SF=1
ADC AL, 2           ;(AL) = (AL) + 2 +(CF)
```

This instruction is useful in performing the addition of 32-bit numbers (i.e., double words) in steps. First the ADD will add the lower words of both the operands and then ADC adds the higher words of both the operands along with carry generated by adding the lower words. In the same manner we can perform higher order additions such as 64-bit additions also.

INC Instruction

The INC instruction is used increment the byte or word operand by 1. The syntax of the INC instruction is as follows:

INC *destination*
(*destination*) ← (*destination*) + 1

The value stored in *destination* is incremented by 1. The *destination* can be register or memory location. All the conditional flags except the carry flag CF are affected. Following are some examples:

```
INC AX                  ;(AX) = (AX) + 1
;increment byte stored at offset SI by 1
INC BYTE PTR [SI]
INC count               ;increment count by 1
```

This instruction works like up-counter. Suppose if the byte to be incremented reached the value FFh, then INC will make it 00h without affecting CF. The FFFFh when incremented will become 0000h.

Subtraction Instructions

The subtraction instructions cover the SUB (subtract), SBB (subtract with borrow), DEC (decrement), CMP (compare) and NEG (negate) instructions. Remember that carry flag works as borrow flag in case of subtract instructions. Let us learn them in detail with examples.

SUB Instruction

The SUB instruction is used to perform the arithmetic subtraction. The syntax for the SUB instruction is as follows:

SUB *destination, source*
(*destination*) ← (*destination*) – (*source*)

The contents of *source* are subtracted from the *destination* and the result is stored in *destination*. The *source* can be immediate value, register or memory location. The *destination* can be register or memory location. Both *source* and *destination* cannot be in memory. All the six conditional flags are updated based on the result stored in *destination*. The carry flag CF works as a borrow flag. Following are some examples:

```
SUB AX, BX                  ;if AX = A236h and BX = 5623h
                            ;result in AX = 4C13h
                            ;OF=1, CF=0, PF=0, AF=0, ZF=0, SF=0
SUB AL, 35h                 ;(AL) = (AL)-35h
SUB SI, WORD PTR [BX]       ;subtract word at offset BX from SI
SUB BYTE PTR [SI], 2        ;reduce the byte at offset SI by 2
SUB AL, n                   ;subtract value of n from AL
```

SBB Instruction

The SBB is known as "subtract with borrow". The syntax for the SBB instruction is as follows.

SBB *destination, source*
(*destination*) ← (*destination*) – (*source*) – (CF)

The contents of *source* and carry flag CF (borrow flag) are subtracted from the *destination* and the result is stored in *destination*. The rest of the things are same as SUB instruction. For example,

```
SBB AX, BX ;(AX) = (AX) - (BX) - (CF)
```

The SBB instruction is useful in carrying out the higher order subtraction in the same way as the ADC instruction. For example, the 32-bit subtraction is done in two steps. First the subtraction of lower words is done and then the higher words are subtracted with borrow flag.

DEC Instruction

The DEC instruction is used to decrement the byte or word operand by 1. The syntax of the DEC instruction is as follows:

```
DEC destination
(destination) ← (destination) – 1
```

The value stored in *destination* is decremented by 1. The *destination* can be register or memory location. All the conditional flags except the carry flag CF are affected. Following are some examples:

```
DEC AX                  ;(AX) = (AX) - 1
;decrement byte stored at offset SI by 1
DEC BYTE PTR [SI]
DEC count               ;decrement count by 1
```

This instruction works like down-counter. Suppose if the byte to be decremented reached the value 00h, then DEC will make it FFh without affecting CF. The 0000h when decremented will become FFFFh.

NEG Instruction

It is known as "negate instruction" which performs 2's complement operation. The syntax for the NEG instruction is as follows:

```
NEG destination
(destination) ← 0 – (destination)
```

The *destination* is replaced by its 2's complement. It is performed simply by subtracting the *destination* from 0. The *destination* can be register or memory location. All the conditional flags are affected. Following are some examples:

```
NEG AL                  ;(AL) = 2'complement of (AL)
                        ;if (AL) = 05h, it is replaced by FBh
NEG WORD PTR [BX]       ;2's complement of word at offset BX
```

The NEG instruction is used to change the sign of the number. In the first example, the original value is 05h which represents +5 in sign-magnitude form. The 2's complement of 05h is FBh which is nothing but –5 in sign-magnitude form.

CMP Instruction

It is known as "compare instruction" and is used to compare two bytes or two words to determine the relation between them – equal, greater than or less than. The syntax for the CMP instruction is as follows:

```
CMP destination, source
(destination) – (source)
```

The *source* is subtracted from the *destination* and all the flags are updated accordingly. Neither of the operands is changed. Assume that the destination is AX register and the source is BX register, then relation between them is determined from the flags as shown in Table 2.

Table 2 Result of CMP AX, BX

Condition	*CF*	*ZF*	*SF*	*Comment*
(AX) = (BX)	0	1	0	;equal as difference is 0, ZF = 1
(AX) > (BX)	0	0	0	;greater than, CF = 0
(AX) < (BX)	1	0	1	;less than, CF = 1 (borrow flag set)

The *source* can be immediate value, register or memory location. The *destination* can be register or memory location. Both *source* and *destination* cannot be in memory. Following are some examples:

```
CMP AL, 32h                ;compare AL with 32h
CMP CX, WORD PTR [SI]      ;compare CX with word at offset SI
CMP BYTE PTR [BX+5], AL    ;compare byte at offset BX+5
                           ;with AL
```

The compare instruction is very useful in decision making as it provides the condition for conditional jump instructions. The compare instruction sets the flags based on the relation between the operands which is/are used by conditional jump instructions to decide the jump to a target instruction. Following is an example:

```
CMP AX, BX      ;compare AX with BX
JC next         ;jump if (AX) < (BX)
```

If (AX) < (BX), the carry flag CF is set because a large number is subtracted from a small number which generates borrow. The JC instruction transfers control to instruction with label `next` as CF = 1.

Unsigned Multiplication and Division Instructions

The 8086 processor provides the instructions for performing multiplication and division of unsigned numbers. The MUL instruction is used for unsigned multiplication and the DIV instruction is used for unsigned division. They are explained as follows.

MUL Instruction

It is known as unsigned multiplication. It multiplies two byte operands or two word operands. The syntax for the MUL instruction is as follows:

```
MUL source
(AX) ← (AL) × (source) for unsigned byte operands and
(DX:AX) ← (AX) × (source) for unsigned word operands
```

The *source* can be register or memory location. If the *source* is an unsigned byte, then it is multiplied with the unsigned byte in the AL register and the 16-bit unsigned result is stored in the AX register with upper byte in AH and lower byte in AL. The multiplication of two 8-bit numbers can go up to 16-bits. If *source* is an unsigned word, then it is multiplied with the unsigned word in the AX register and the 32-bit unsigned result is stored in the 32-bit extended register DX:AX with upper word in DX and lower word in AX. The multiplication of two 16-bit numbers can go up to 32-bits. If the

upper byte of a 16-bit result or upper word of a 32-bit result contains all the 0 bits, then CF = OF = 0, otherwise CF = OF = 1. This means that if part of the result is stored in upper byte or upper word, then both CF and OF are set. This information is useful to decide whether or not we can discard upper byte or upper word from the result. The conditional flags AF, PF, ZF and SF are undefined for this instruction. Following are some examples:

```
MUL BL                      ;if BL = 5, AL = 10, then AX =32h
                            ;(50 decimal)
MUL WORD PTR [BX+SI]        ;multiply word at offset BX + SI
                            ;with AX, result in DX:AX
```

If you need to multiply an unsigned byte with an unsigned word, then first you need to convert an unsigned byte to an unsigned word by putting 0s in upper 8-bits and then multiply two words. Consider a following example:

```
MOV AL, 05h      ;load AL with unsigned value 05h
MOV BX, 2312h    ;load AX with unsigned value 2312h
MOV AH,00h       ;convert to unsigned word AX = 0005h
MUL BX           ;multiply
```

Program 3.9

Write a program to perform scalar multiplication of array of five unsigned bytes.

Solution

The program is

```
data    SEGMENT
        n      DB 03h
        val    DB 12h,0A5h,0FFh,34h,98h
        s_val  DW 5 dup(0)
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data                ;initialize the data segment
        MOV DS, AX

        MOV CX, 5                   ;load the count
        MOV BL, n                   ;get scalar constant
        LEA SI, val                 ;pointer to original values
        LEA DI, s_val               ;pointer to scaled values
next:   MOV AL, BYTE PTR [SI]       ;read next value
        MUL BL                      ;multiply by scalar constant
        MOV WORD PTR [DI], AX       ;store scaled value
        INC SI                      ;forward pointer
```

```
        ADD DI, 2               ;forward pointer
        LOOP next               ;go to next

        MOV AX, 4C00h           ;terminate the program
        INT 21h

code    ENDS
        END start
```

As seen in Program 3.9, each time in a loop, a byte value pointed by SI is transferred to AL, multiplied by the constant in BL and then stored at the next word position pointed by the DI. The pointer SI is incremented by 1 to point to next byte and the pointer DI is incremented by 2 to point to next word position.

DIV Instruction

It is known as unsigned division and used to divide an unsigned word by an unsigned byte or unsigned double word by an unsigned word. The syntax for the DIV instruction is as follows:

```
DIV source
(AH) = reminder, (AL) = quotient ← (AX) / (source byte)
(DX) = reminder, (AX) = quotient ← (DX:AX) / (source word)
```

The *source* can be register or memory location. If the divisor is an unsigned byte, then the dividend must be an unsigned word in the AX register. After performing division operation, the 8-bit remainder will be stored in the AH register and the 8-bit quotient will be stored in the AL register. If the divisor is an unsigned word, then the dividend must be unsigned double word in the DX:AX register. After performing division operation, the 16-bit remainder will be stored in the DX register and the 16-bit quotient will be stored in the AX register. The quotient will always be truncated. All the conditional flags are undefined for the DIV instruction. If we try to divide by 0 or if the quotient does not fit in the destination (greater than FFh or FFFFh) then the interrupt type 0 known as "divide by zero" will be called. We will learn the 8086 interrupts in Chapter 7.

For performing division of unsigned byte by unsigned byte or unsigned word by unsigned word, the dividend must be extended by putting 0s in the upper byte or upper word and then division by word/byte or double word/word can be used. Following are some examples.

```
DIV BL                      ;if AX=0033h (51 decimal) and BL=2,
                            ;the remainder AH=1 and
                            ;question AL =19h (25 decimal)
DIV WORD PTR  [SI]          ;divide DX:AX by word at offset SI
```

Program 3.10

Write a program to divide 32-bit unsigned number by an 16-bit unsigned number.

Solution

The program is

```
data    SEGMENT
        dividend  DD   11204534h
```

```
          divisor   DW   1250h
          rem       DW    ?
          quetient  DW    ?
data      ENDS
code      SEGMENT
          ASSUME cs:code, ds:data
start:    MOV AX, data      ;initialize the data segment
          MOV DS, AX
          LEA BX, dividend ;get the offset of dividend
          MOV AX, WORD PTR [BX]    ;load AX with lower word
          MOV DX, WORD PTR [BX+2] ;load DX with upper word
          DIV divisor              ;divide by divisor
          MOV rem, DX              ;store remainder
          MOV quetient, AX         ;store quetient
          MOV AX, 4C00h     ;terminate the program
          INT 21h
code      ENDS
          END start
```

Signed Multiplication and Division Instructions

The 8086 also provides instructions for multiplication and division for the signed operands, namely, IMUL and IDIV. They work in the same way as their unsigned versions. Before learning them, let us first learn the instructions, converting signed byte to signed word CBW and converting signed word to signed double word CWD useful to extend the signed operands. Remember that signed numbers are stored normally in sign-magnitude form using 2's complement.

CBW Instruction

This instruction converts the signed byte stored in the AL register to a signed word by copying the sign bit of AL into all the bits of AH. Consider the following examples:

```
CBW       ;if AL = 0000 0101 (+5),
          ;then AX = 0000 0000 0000 0101 (+5)
CBW       ;if AL = 1111 1011 (-5)
          ;then AX = 1111 1111 1111 1011 (-5)
```

CWD Instruction

This instruction converts the signed word stored in the AX register to a signed double word by copying the sign bit of AX into all the bits of DX. Consider the following example:

```
CWD       ;if AX = 1100 1111 0001 1111  (-12513)
          ;then (DX:AX) = 1111 1111 1111 1111 1100 1111
          ;0001 1111 (-12513)
```

These instructions are useful for extending signed operands while performing signed multiplication and division operations. Let us learn IMUL and IDIV instructions to perform the signed multiplication and signed division.

IMUL Instruction

It is known as signed multiplication. The syntax for the IMUL instruction is as follows:

```
IMUL source
(AX) ← (AL) × (source) for signed byte operands and
(DX:AX) ← (AX) × (source) for signed word operands
```

The *source* can be register or memory location. If *source* is a signed byte, then it is multiplied with the signed byte in the AL register and the 16-bit signed result is stored in the AX register. If *source* is a signed word, then it is multiplied with the signed word in the AX register and the 32-bit signed result is stored in the 32-bit extended register DX:AX. If the magnitude of the result does not need all the bits of the destination (AX or DX:AX), then rest of the upper bits are filled with sign bit 0 or 1 of the result. If AH in case of 16-bit result and DX in case of 32-bit result contain only the copy of sign at all the bit positions then CF = OF = 0, otherwise CF = OF = 1 and they contain part of the result. The rest of the conditional flags are undefined for the IMUL instruction. Assuming that AL = 32h (+50 decimal) and BL = 15h (+21), the result of IMUL is as follows.

```
IMUL BL  ;result AX = 041Ah (+1050)
         ;CF = OF = 1
```

If we change the AL to CEh (–50) then the result is

```
IMUL BL  ;result AX = FBE6h (-1050)
         ;CF = OF = 1
```

If we need to multiply a signed byte by a signed word, then first signed byte is to be converted to signed word by using CBW instruction and then we can multiply two signed words.

IDIV Instruction

It is known as "signed division". The format of the IDIV instruction is as follows.

```
IDIV source
(AH) = signed reminder, (AL) = signed quotient ← (AX) / (source byte)
(DX) = signed reminder, (AX) = signed quotient ← (DX:AX) / (source word)
```

The *source* can be register or memory location. If the divisor is a signed byte, then the dividend must be a signed word in the AX register. After performing division operation, the 8-bit signed remainder will be stored in the AH register and the 8-bit signed quotient will be stored in AL register. If the divisor is a signed word, then the dividend must be signed double word in the DX:AX register. After performing division operation, the 16-bit signed remainder will be stored in the DX register and the 16-bit signed quotient will be stored in the AX register. The sign of the remainder is same as dividend and the quotient will always be truncated. All the conditional flags are undefined for the IDIV instruction. If we try to divide by 0 or if the quotient is out of range (–128 to +127 for 8-bit and –32768 to

+32767 for 16-bit), then the interrupt type 0 – known as "divide by zero" – will be called. We will learn the 8086 interrupts in Chapter 7. Following is an example:

```
IDIV BL  ;if AX = FBE6h (-1050), BL = 07h (+7)
         ;result into 'Divide overflow'
```

To divide signed byte by signed byte, first convert dividend into signed word using CBW and to divide signed word by signed word, convert dividend into signed double word using CWD.

Check Points

We now know that

- the arithmetic instructions contain instructions to perform arithmetic operations add, subtract, multiply and divide operations.
- addition instructions include ADD, ADC and INC instructions.
- subtract instructions include SUB, SBB, DEC, NEG and CMP instructions.
- multiply instructions include MUL instruction for unsigned multiplication and IMUL instruction for signed multiplication.
- division instructions include instruction for signed extension of operand CBW and CWD, DIV instruction for unsigned division and IDIV instruction for signed division.

3.8 Logical Instructions

The logical instructions are used to perform the bit-wise logical operations on byte or word operands. This group includes the instructions for performing bit-wise operations including AND, OR, XOR and NOT. It has one more instruction called TEST which is same as AND but updates only flags without changing operands. Let us learn all of them in detail.

AND Instruction

The AND instruction performs the bit-wise AND operation. The syntax for the AND instruction is as follows:

```
AND destination, source
(destination) ← (destination) & (source)
```

The symbol & denotes the bit-wise AND operation. The contents of `source` and `destination` are bit-wise ANDed and the result is stored in `destination`. The resulting bit is 1 if both the bits at same positions in `source` and `destination` are 1, otherwise the resulting bit is 0. This is same as the operation of 2-input logical AND gate. The `source` can be immediate value, register or memory location. The `destination` can be register or memory location. Both `source` and `destination` cannot be in memory. The CF and OF are cleared after execution of the instruction. The SF and ZF are updated based on the result of AND operation. AF is undefined for this operation. PF is only defined for the byte and updated based on only lower 8-bits of the result.

The AND operation is very useful for masking. Following are some examples:

```
AND AX, BX          ;if AX = 0000 0101 0111 1101 and
                    ;BX = 0111 0111 0100 1001 the result
```

```
                        ;AX = 0000 0101 0100 1001
                        ;CF = OF = 0, SF = 0, ZF = 0, PF = 0
AND AX, 00FFh           ;Mask the upper byte ox AX without
                        ;changing lower byte
AND BYTE PTR [SI], AL   ;AND byte at offset SI with AL
AND AX, WORD PTR [BX]   ;AND AX with word at offset BX
```

OR Instruction

The OR instruction performs the bit-wise OR operation. The syntax for the OR instruction is as follows:

```
OR destination, source
(destination) ← (destination) | (source)
```

The symbol | denotes the bit-wise OR operation. The contents of *source* and *destination* are bit-wise ORed and the result is stored in *destination*. The resulting bit is 1 if either of the bits or both the bits at same positions in *source* and *destination* are 1, otherwise the resulting bit is 0. This is same as the operation of 2-input logical OR gate. The *source* can be immediate value, register or memory location. The *destination* can be register or memory location. Both *source* and *destination* cannot be in memory. The CF and OF are cleared after execution of the instruction. The SF and ZF are updated based on the result of OR operation. AF is undefined for this operation. PF is only defined for the byte and updated based on only lower 8-bits of the result.

The OR operation is useful to set the desired bits to 1 in the operand. Following are some examples:

```
OR AL, 0Fh              ;if AL = 0101 0100, the AL | 0000 1111
                        ;gets the result AL = 0101 1111
                        ;CF = OF = 0, SF = 0, ZF = 0, PF = 1
OR AX, WORD PTR [SI]    ;OR the AX with word at offset SI
OR AX, BX               ;perform (AX) | (BX)
```

NOT Instruction

The NOT instruction performs the bit-wise 1's complement operation. The syntax for the NOT instruction is as follows:

```
NOT destination
(destination) ← ~(destination)
```

The symbol ~ denotes the bit-wise 1's complement. The contents of the *destination* are complemented, that is, bits are inverted and stored at the same place. This is same as the operation of Logical NOT gate. The *destination* can be register or memory location. This instruction does not affect the flags. Following are some examples:

```
NOT AX                  ;if AX = 0011 1010 1100 1101 then result
                        ;in AX = 1100 0101 0011 0010
NOT WORD PTR [SI]       ;1's complement of word at offset SI
```

Program 3.11

Write a program to perform ORing of two 16-bit numbers without using OR instruction. [Hint: De Morgan's rule, $A + B = (A' \cdot B')'$]

Solution

The program is

```
data    SEGMENT
        n1      DW      1234h
        n2      DW      4321h
        ans     DW        ?
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data        ;initialize the data segment
        MOV DS, AX

        MOV AX, n1          ;load AX with n1
        NOT AX              ;complement AX
        MOV BX, n2          ;load BX with n2
        NOT BX              ;complement BX
        AND AX, BX          ;AND AX with BX
        NOT AX              ;complement result
        MOV ans, AX         ;store result

        MOV AX, 4C00h       ;terminate the program
        INT 21h

code    ENDS
        END start
```

As seen in Program 3.11, the operands are complemented using NOT instruction, then the ANDing of complemented operands is performed and finally the result of AND operation is complemented to get the ORing of operands.

XOR Instruction

The XOR instruction performs the bit-wise XOR operation. The syntax for the XOR instruction is as follows:

```
XOR destination, source
(destination) ← (destination) ^ (source)
```

The symbol ^ denotes the bit-wise XOR operation. The contents of `source` and `destination` are bit-wise XORed and the result is stored in `destination`. The resulting bit is 1 if either of

the bits at same positions in *source* and *destination* is 1, otherwise it is 0. This is same as the operation of the logical EX-OR gate. The *source* can be immediate value, register or memory location. The *destination* can be register or memory location. Both *source* and *destination* cannot be in memory. The CF and OF are cleared after execution of the instruction. The SF and ZF are updated based on the result of XOR operation. AF is undefined for this operation. PF is only defined for the byte and updated based on only lower 8-bits of the result.

The XOR operation is useful to generate and check the parity used for handling the 1-bit errors. Following are some examples:

```
XOR AX, BX                    ;if
                              ;AX=1000 0101 0111 1101 and
                              ;BX=0111 0111 0100 1001 the result
                              ;AX=1111 0010 0011 0100
                              ;CF=OF=0, SF=1, ZF=0, PF=0
XOR BYTE PTR [BX+2], AL       ;XOR byte at offset BX+2 with AL
```

TEST Instruction

The TEST instruction performs the bit-wise AND operation to update the flags without changing operands. The syntax for the TEST instruction is as follows:

```
TEST destination, source
(destination) & (source) ;flags are updated
```

The contents of *source* and *destination* are bit-wise ANDed to update the flags based on the result. Neither of the operands is changed. The *source* can be immediate value, register or memory location. The *destination* can be register or memory location. Both *source* and *destination* cannot be in memory. The CF and OF are cleared after execution of the instruction. The SF and ZF are updated based on the result of AND operation. AF is undefined for this operation. PF is only defined for the byte and updated based on only lower 8-bits of the result. Following is an example of TEST instruction.

```
TEST AL, CL    ;if AL=1010 1010 and CL=0000 0000
               ;flags CF=OF=0, SF=0,  ZF=1, PF=1
```

Check Points

We now know that

- the logical instructions perform the bit-wise operations on the operands similar to logic gates.
- the instruction in this group are AND, OR, NOT, XOR and TEST.
- the AND and TEST both perform bit-wise ANDing, the only difference is that TEST does not change operands.

We have learned data transfer, arithmetic and logical instructions and used them in simple programs, mostly sequential. Once we learn branching and looping in the next chapter, we will use all these instructions to write complex and more meaningful programs.

Summary

- Assembly language program contains the 8086 instructions and pseudo-instructions. The 8086 instructions are translated to machine codes. The pseudo-instructions are called assembler directives and do not occupy any space in machine codes.
- The assembler directives are used to direct the assembler to correctly and efficiently translate the programs. The basic directives are commonly used directives.
- The 8086 assembly language program consists of one or more logical segments created using any text editor and stored as *.asm* file.
- The assembler like Microsoft's Macro Assembler (MASM) accessed as "masm" is used to assemble the program which is linked using linker program "link" to get the executable file *.exe*.
- The .exe file is executed like any DOS command.
- To see the results in internal registers and memory locations, a tool called debugger is used. It is available in the DOS as "debug command.
- Debug also provides the facility to run progra in step mode to identify the logical errors.
- The DOS itself is build around the interru services which can be called in our progra using INT instruction. The INT 21h is th major DOS service. Functions 1 and 2 service 21h provide character I/O where functions 9 and 10 provide reading an writing a string.
- The data transfer instructions are used t move the data between registers, register an memory, register and I/O ports.
- The arithmetic instructions are used t perform the arithmetic operations. The 808 provides arithmetic instructions for both binary and BCD numbers.
- The logical instructions are used to perform the bit-wise logical operations.

Glossary

Assembler directives are pseudo-instructions used to direct the assembler for translation.

Logical segment is a segment written in the 8086 assembly language program using SEGMENT and ENDS directives.

Physical segment is a block of computer memory pointed by segment register and stores a logical segment in machine form.

MASM is a Macro Assembler from Microsoft used to assemble the assembly language program.

Object program is a machine code containing unresolved references.

Linker is a program which resolves the references in the object program and produces the executable code.

Debugger is a software tool used to debug and run the programs in a step mode.

Debug is DOS command for debugging.

DOS Service is an Interrupt Service Routine (ISR) providing number of functions.

Objective Questions

State whether true/false. Give reason for your answer

1. Each statement in an assembly language program occupies space in machine language program.
2. One physical segment is pointed only by one segment register.
3. A segment written in assembly language program is a logical segment.
4. The code segment register is automatically initialized when program is loaded while data segment is not.
5. The label used with END directive is always a label to first instruction in code segment.
6. Debug allows us to see the contents of internal register but does not allow us to modify them.
7. The maximum number of functions that can be provided by a DOS service is 255.
8. We cannot call the DOS service in our program.
9. A string in DOS must be ended with '#' sign to print it properly.
10. The maximum length of an input string in DOS is 255 characters.
11. The arithmetic instructions work only with binary numbers.
12. We cannot multiply a byte and a word using MUL instruction.
13. The TEST instruction performs the logical AND operation but does not change operands.
14. Dividing a number by 0 results into "Divide Overflow".
15. The NEG instruction changes the sign of an operand.

Multiple-Choice Questions

1. What is an extension of the file produced by linker?
   a. .asm
   b. .obj
   c. .exe
   d. None of the above
2. Which of the following debug command allows you to see the memory contents?
   a. u – unassemble
   b. e – enter
   c. r – register
   d. t – trace
3. Which of the following directive gives the segment address of a variable?
   a. SEGMENT
   b. PTR
   c. SEG
   d. All of the above
4. Which of the following instruction is invalid?
   a. ADD AX, BX
   b. ADD AX, [SI]
   c. ADD [SI], [BX]
   d. ADD AX, WORD PTR [SI]

5. A variable defined using DQ occupies
   a. 16 bits
   b. 32 bits
   c. 64 bits
   d. 80 bits

6. Which of the following instruction is not a logical instruction?
   a. NEG
   b. NOT
   c. XOR
   d. TEST

7. What will be the value of AX after executing CBW with AL = 82h?
   a. 0082h
   b. FF82h
   c. 8282h
   d. None of the above

8. What is true for the signed division?
   a. Quotient is truncated
   b. Sign of reminder is same as dividend
   c. Both
   d. None

Review Questions

1. What are assembler directives? What is their role? Why are they called pseudo-instructions?
2. List the various assembler directives and explain any three of them.
3. How do you define a segment? What is the role of ASSUME directive?
4. Explain various data definition directives with examples.
5. Show the use of SEG and OFFSET directives.
6. What is the use of PTR directive?
7. How does assembler know the end of the program?
8. How do you assemble and link the program stored in .asm file? Explain with example.
9. What do you mean by debugging of a program?
10. List the various debug commands with their meaning.
11. What is DOS service? How can you access it?
12. Explain the display of a string function of service 21h.
13. Give and explain the format of buffer for input string.
14. Explain the working of XLAT instruction.
15. Why 8086 instructions using two operands do not allow both of them in memory?
16. List the address transfer instructions and explain their operations.
17. Explain the working of CMP instruction.
18. Differentiate between AND and TEST instructions.
19. Explain the working of MUL and DIV instructions.
20. Explain the working of IMUL instruction with example.
21. Explain the working of CBW and CWD instructions with example.

22. Find the errors in following instructions and correct them.
    a. MOV AL, BX
    b. MOV 1234h, AX
    c. MOV CS, 2345h
    d. MOV [SI], [BX]
    e. INC [SI]
    f. ADD [3400h], AL
    g. DEC 12h
    h. MUL 12h
    i. DIV [SI+2]

23. Write an instruction(s) to perform the following operations.
    a. Loading 45A7h into AX register.
    b. Increment the contents of memory location whose offset address is 3412h by1.
    c. Set the lower four bits of AX register to 1 without changing other bits.
    d. Make the upper byte of a word at offset SI to 0 without changing lower byte.
    e. Multiply 12h with AX.
    f. Divide AX by BX.

Programming Problems

1. Write a program to print your address.
2. Write a program to read a string from keyboard and print it in uppercase letters only.
3. Write a program to add two 32-bit numbers.
4. Write a program to subtract a byte from a word.
5. Write a program to perform ANDing of two words without using AND instruction.
6. Write a program to compute checksum of five words using XOR instruction.
7. Write a program to divide a 32-bit number by 16-bit number.

Answers

True or False. Reasons are left as an exercise

1. False
2. False
3. True
4. True
5. False
6. False
7. True
8. False
9. False
10. True
11. False
12. False
13. True
14. True
15. False

Multiple-Choice Questions

1. (c)
2. (b)
3. (c)
4. (c)
5. (c)
6. (a)
7. (b)
8. (c)

4 Branching and Looping

LEARNING OBJECTIVES

At the end of this chapter, you will be able to:

- Describe the unconditional jump instruction and use it in programs.
- Define short and near jump.
- Differentiate between intrasegment jump and intersegment jump.
- List and explain the various conditional jump instructions and use them in programs.
- Describe the looping instructions and use them in programs.
- List and explain the instructions to perform the ASCII and BCD arithmetic and write the programs using them.
- List and define the processor control instructions.

4.1 Introduction

In the last chapter, we have written simple programs using data transfer, arithmetic and logical instructions. Most of the programs were sequential in nature, that is, they are executed from the first to last instruction in linear manner. The sequential programs are not useful to solve real-life problems. To solve real-life problems, decision making and looping are must. The 8086 provides the unconditional and conditional jump instructions for that purpose. The unconditional jump is used for branching to any instruction within a program without any condition. The 8086 provides different types of unconditional jump instructions like short and near, and far, which are also called *intrasegment* and *intersegment*, respectively. The conditional jump instructions use conditional flags as conditions. If the condition is in favor, they take the jump to target instruction, otherwise they move to the next instruction in sequence.

The 8086 microprocessor supports special iteration control instructions, also known as *looping instructions*, for efficient looping. These instructions implicitly use the CX as count register to control the iterations. The 8086 microprocessor supports the processing not only of binary numbers but also of ASCII and Binary Coded Decimal (BCD) numbers using special arithmetic instructions. The processor control instructions are used to change or control the way the processor behaves. These include flag set/reset instructions, no operation instruction and external hardware synchronization instructions.

4.2 Unconditional Jump Instruction

Much of the power of programming comes from the branch control instructions which allow transfer of control arbitrarily anywhere within the program. The unconditional and conditional jump instructions, provided in the 8086 instruction set, fall in this category. The unconditional

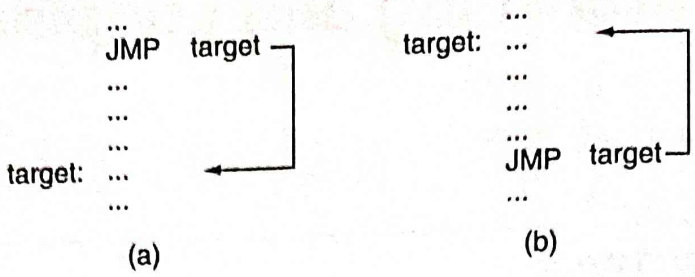


Figure 1 (a) Forward jump and (b) backward jump.

jump instruction takes the jump unconditionally, that is, it always jumps once execute, whereas th conditional jump instruction jumps only if the specified condition is true, otherwise it simply act as no operation (NOP) and the control moves to the next instruction after it. The jump, whethe conditional or unconditional, can be categorized into forward jump or backward jump dependin on the position of the target instruction with reference to the position of the jump instruction itself Figure 1 shows the structure of the forward and backward jumps using the unconditional jum instruction JMP.

As can be seen from Fig. 1(a), the target instruction is after the jump instruction in the forwar jump. While the jump instruction is being executed, the Instruction Pointer points to the next instruc tion after the JMP instruction. The JMP instruction causes the change of IP value to the IP of the targe instruction, that is, the instruction containing the label mentioned in the jump instruction. Hence the next instruction fetched after the JMP instruction is the instruction with label `next` as the IP i changed to the IP of instruction with label `next`. This is known as transferring the execution control In the case of the backward jump, as shown in Fig. 1(b), the target instruction comes before the jump instruction. It also causes the transfer of control in the same manner as in the forward jump by chang ing the IP. The concept also applies to the conditional jump instructions which are discussed in the nex section. This section focuses on the various types of unconditional jump instructions.

The physical address of an instruction is computed by adding the offset or displacement in the IP register to the segment base in the CS register. If the target instruction and the jump instruction both are in the same segment then their offsets are different but the segment base is same. This means that to perform the jump, we need to change only the IP value. If the target instruction and the jump instruc tion are in different segments then to perform the jump, we need to change both CS and IP values a their segment bases and offsets both are different. The 8086 unconditional jump instruction JMP is further categorized into intrasegment (short or near) and intersegment (far) based on whether only IP is changed or IP and CS both are changed.

Intrasegment (Near or Short) Jumps

Intrasegment means within the segment, that is, the target instruction and the jump instruction both are in the same segment. This means that we need to change only the IP value in order to perform the jump. The intrasegment jump is of two types:

1. Direct.
2. Indirect.

The intrasegment direct instruction specifies the offset of the target instruction as displacement directly in the instruction, whereas intrasegment indirect instruction specifies the offset of the target instruction

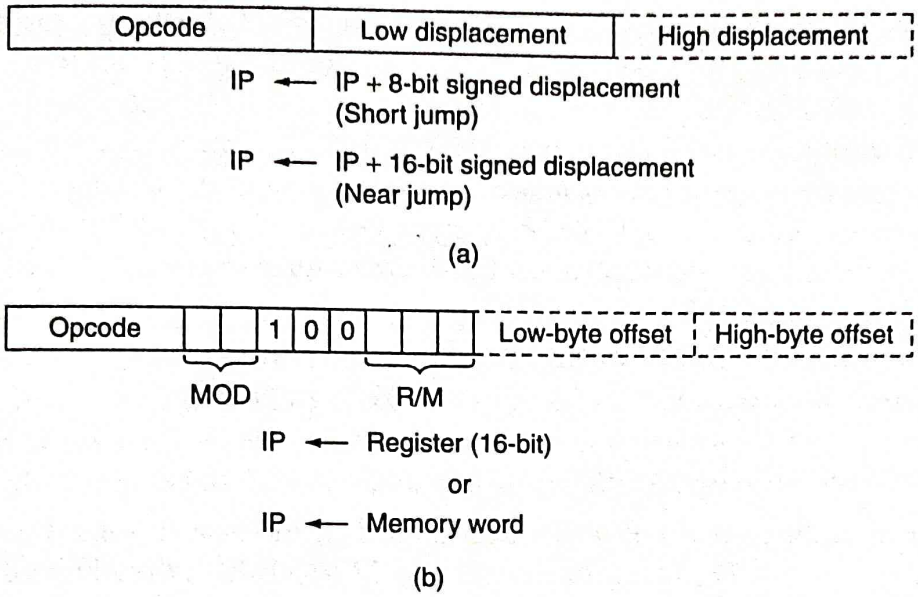


Figure 2 Intrasegment (a) direct, IP relative and (b) indirect, JMP instructions.

using 16-bit register or memory word indirectly. Figure 2 shows the format of the intrasegment direct and indirect JMP instructions.

As can be seen from Fig. 2(a), the intrasegment direct jump contains 2 or 3 bytes, 1 byte operation code (opcode) followed by 8-bit or 16-bit signed displacement. The third byte of the instruction is used only if the displacement is 16-bit. The displacement shows the distance of the target instruction in bytes from the instruction that is next to the JMP instruction. If the jump is forward, the displacement is positive, and if the jump is backward, the displacement is negative. The IP of the target instruction is computed by simply adding the 8-bit (sign extended to 16-bit) or 16-bit displacement in the current value of IP, that is, the offset address of the next instruction after the JMP. If the displacement is 8-bit, the range of jump in both the directions, that is, backward and forward, is −128 to +127 bytes from the IP of the next instruction, and it is known as the *short jump*. If the displacement is 16-bit, the range of jump in both the directions, that is, backward and forward, is −32768 to +32767 bytes from the IP of the next instruction, and it is known as the *near jump*. The near jump covers the full segment as the size of segment in the 8086 is 64 KB. Let us see how the intrasegment direct instruction works. Consider the following code:

```
        MOV CX, 5
next:   ADD AX, BX
        NOP
        NOP
        JMP next
```

The binary coding of the above code looks like

```
1813:0000 B90500        MOV CX, 5
1813:0003 03C3          ADD AX, BX
1813:0005 90            NOP
1813:0006 90            NOP
1813:0007 EBFA          JMP 0003
1813:0009 ED
```

As shown in the above code, the jump instruction is coded as EBFAh where the first byte EBh is the opcode and the second byte FAh (decimal –6) is the 8-bit signed displacement from the IP of the next instruction after JMP. The IP of the instruction after JMP is 0009h and the IP of the instruction with label next, that is, the target instruction ADD, is 0003h. Hence, the distance between them is 6 bytes. The above is the backward jump, hence the displacement is negative, that is, –6 (FAh). When the JMP instruction is being executed, the IP register contains 0009h, and to compute the IP of the target instruction, the 8-bit signed displacement FAh after sign extension to 16-bit (FFFAh) is added to current IP value, that is, 0009h, as

```
    0000 0000 0000 1001 (0009h)
+   1111 1111 1111 1010 (FFFAh)
    -------------------------
    0000 0000 0000 0011 (0003h)
```

The result after discarding the overflow is 0003h which is the new IP value from which the next instruction is to be fetched. The instruction at the new IP (0003h) is the ADD instruction which is the target instruction for the JMP instruction.

The forward short JMP instruction works in the same manner as the backward short JMP instruction except that in the case of the former the displacement to be added to the current IP is positive. The near jump, whether forward or backward, works exactly in the same manner as short jumps with the only difference being that the displacement to be added to the current IP is 16-bit signed value.

Remember, in the case of the backward jump, when the assembler assembles the JMP instruction, the distance (i.e., the displacement to the target instruction) is known as the target instruction comes before the jump instruction. This enables the assembler to decide whether the JMP instruction is coded as 2 or 3 bytes, that is, short or near jump. If the distance is less than or equal to –128 bytes, it is coded with the 8-bit signed displacement (i.e., 2 bytes) opcode followed by 8-bit displacement (short jump), otherwise it is coded with the 16-bit displacement (i.e., 3 bytes) opcode followed by 16-bit displacement (near jump). The distance to the target instruction is not known in the case of the forward jump while coding the JMP instruction, as the target instruction comes later. This causes the problem to the assembler in coding the JMP instruction as short or near, that is, to reserve the 1 byte or 2 bytes for the displacement in addition to 1 byte for opcode. In this case the assembler always reserves 2 bytes for the displacement. When it encounters the target instruction and the displacement is less than or equal to +127 bytes (i.e., 8-bit), then it uses the first reserved byte for displacement (coded as short jump) and the second reserved byte is converted to NOP instruction. If the displacement computed is 16-bit, then both the reserved bytes are used for storing the 16-bit displacement (coded as near jump). If the forward jump is coded as short jump, then the additional instruction NOP is inserted to fill the gap, which may affect the execution time of the program. If you are sure that the distance is in range of 8-bit, then the jump can be written with the assembler directive SHORT as follows:

```
JMP SHORT next
```

This will always code the JMP instruction as short jump (i.e., 2 bytes code), thus solving the problem of performance.

Another point to be kept in mind about the near jump instruction is that if the near jump instruction with the forward jump is placed near the end of the segment and if there are no sufficient bytes left between the jump and the end of the segment, the assembler automatically takes the rest of the distance downward from the start of the segment. This is also true when the near jump instruction with the backward jump is placed near the start of the segment. In this case, the assembler automatically takes the rest of the distance upward from the end of the segment.

The intrasegment direct jump instruction, short or near, is also known as *IP relative* as the instruction uses displacement rather than the absolute value to compute the IP of the target instruction. This allows us to write the relocatable programs which can be loaded anywhere in the memory or relocated even after loading without any problem. This is why the intrasegment direct jump instruction is the most widely used instruction as compared to other unconditional jump instructions. All the jump instructions used in programming examples in next section fall in this category.

Figure 2(b) shows the format of intrasegment indirect instruction. It uses either the 16-bit register or a word in memory to specify the offset of the target instruction using 8 register modes or 24 memory addressing modes. Once such instruction is executed, the 16-bit offset value in the register or memory, depending on the addressing mode used, is loaded into the IP register and the control is transferred to the instruction located at that IP. As the offset value specified in the register or memory is 16-bit, the range of the intrasegment indirect instruction is −32768 to +32767 bytes from the IP of the next instruction. Thus, it falls into the near jump instruction category. For example,

```
JMP BX          ;(IP) ← (BX)
```

The above instruction jumps to an instruction whose offset is specified as content of the BX register. The instruction

```
JMP WORD PTR [BX]        ;(IP) ← (DS:BX)
```

jumps to an offset stored in the data segment at the offset BX.

Intersegment (Far) Jumps

Intersegment means "between two segments." This means that the target instruction and the jump instruction both are in different segments, and the jump needs to change both the segment base in CS and the offset in IP. It is also known as far jump as the control is transferred out of the segment in which the jump instruction resides. The intersegment jump is also of two types:

1. Direct.
2. Indirect.

The intersegment direct jump specifies the offset and the segment address of the target instruction directly in the instruction, whereas the intersegment indirect jump specifies the offset and the segment address of the target instruction indirectly in memory as double word. Figure 3 shows the format of the intersegment direct and intersegment indirect jump instructions.

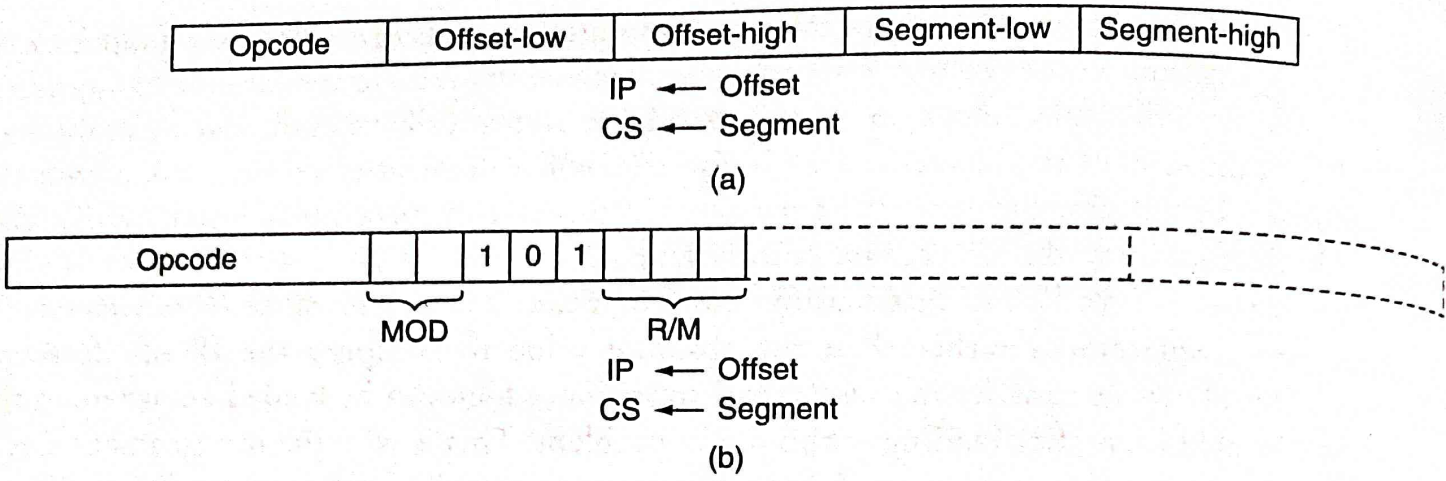


Figure 3 Intersegment (a) direct and (b) indirect jump instructions.

As can be seen from Fig. 3(a), the intersegment direct jump instruction consists of 5 bytes – 1 byte opcode followed by 2 bytes for offset and 2 bytes for segment. Once it executes, the IP is loaded with the offset value stored in the second and third bytes, and CS is loaded with the fourth and fifth bytes and the next instruction is fetched from the new address formed by new values in CS:IP, that is, at the offset specified by the IP register in the segment whose segment base is specified by the CS register.

Figure 3(b) shows the format of the indirect intersegment jump instruction. It uses the 24 memory addressing modes to specify the offset and the segment address of the target instruction. Assume that the following statement defines a double word in data segment:

```
addr  DD 0ffff0000h
```

The following instruction jumps to an instruction at offset `0000h` in the segment with segment base `FFFFh`:

```
MOV SI, OFFSET addr      ;get offset of addr
JMP DWORD PTR [SI]       ;far jump to FFFF:0000h
```

First, the instruction loads the offset of variable `addr` in the SI register, and then the JMP instruction makes the far jump at the address pointed by the SI register. The first word pointed by SI is loaded in IP and the next word is loaded in CS. Hence the IP = `0000h` and the CS = `FFFFh` transferring control to `FFFF:0000h`.

Considering the above discussion for the unconditional jump instructions and the conditional jump instructions in the next section, we can classify the jump instructions of the 8086 microprocessor as shown in Fig. 4. Remember, none of the jump instructions affect the flags.

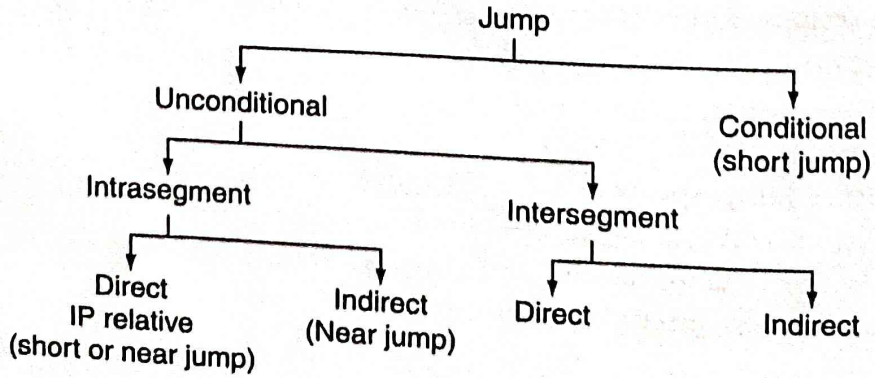


Figure 4 Classification of the 8086 jump instructions.

Check Points

We now know that

- the unconditional jump takes the jump unconditionally. The 8086 microprocessor provides the JMP instruction for unconditional jump.
- the forward jump contains the target instruction after the jump instruction, whereas the backward jump contains the target instruction before the jump instruction.
- the intrasegment jump needs to change only IP, as both jump and target instructions are in the same segment.
- the intrasegment jump can be direct or indirect. The intrasegment direct jump, also known as IP relative, is either short or near jump. The intrasegment indirect jump is always near.
- the range of the short jump is −128 to +127 bytes from the address of the next instruction, and for the near jump is −32768 to +32767 bytes.
- the intersegment jump needs to change both CS and IP, as target and jump instructions are in different segments. The intersegment jump is also of direct and indirect types.

4.3 Conditional Jump Instructions

The 8086 provides six conditional flags:

1. Carry flag (CF).
2. Auxiliary carry flag (AF).
3. Parity flag (PF).
4. Zero flag (ZF).
5. Sign flag (SF).
6. Overflow flag (OF).

These flags are set/reset as a result of execution of normally arithmetic and logical instructions. The conditional jump instructions perform jump to a target instruction only if the specified condition is true, otherwise the control moves to the next instruction after the conditional jump instruction. How do we form the conditions for the conditional jump instructions? The answer is that the status of the conditional flags forms the conditions for the conditional jump instructions, and the jump is decided based on whether the condition in terms of flag(s) status is true or false. For example, for the instruction JC (Jump if Carry), the carry flag provides the condition. If carry flag CF = 1, the condition is true and jump is performed, otherwise the condition is false and the control moves to the next instruction after the instruction JC. Table 1 lists all the conditional jump instructions provided by the 8086 microprocessor.

As can be seen from Table 1, the first 10 instructions are based on the five conditional flags except the auxiliary carry flag (AF). These instructions jump on the status of a single conditional flag. For example, JZ jumps if ZF = 1 and JNZ jumps if ZF = 0. Normally, the conditional jump instructions are used after the arithmetic and logical instructions, as the arithmetic and logical instructions update the conditional flags based on their result. Consider the following code snippet:

Table 1 Conditional jump instructions

Instruction	*Flags used*	*Function*
JC	CF = 1	Jump if carry
JNC	CF = 0	Jump if not carry
JE/JZ	ZF = 1	Jump if equal/zero
JNE/JNZ	ZF = 0	Jump if not equal/zero
JO	OF = 1	Jump if overflow
JNO	OF = 0	Jump if not overflow
JP/JPE	PF = 1	Jump if parity/even parity
JNP/JPO	PF = 0	Jump if not parity/odd parity
JS	SF = 1	Jump if sign
JNS	SF = 0	Jump if not sign
JA/JNBE	(CF OR ZF) = 0	Jump if above/not below or equal
JAE/JNB	CF = 0	Jump if above or equal/not below
JB/JNAE	CF = 1	Jump if below/not above or equal
JBE/JNA	(CF OR ZF) = 1	Jump if below or equal/not above
JG/JNLE	((SF XOR OF) OR ZF) = 0	Jump if greater/not less or equal
JGE/JNL	(SF XOR OF) = 0	Jump if greater or equal/not less
JL/JNGE	(SF XOR OF) = 1	Jump if less/not greater or equal
JLE/JNG	((SF XOR OF) OR ZF) = 1	Jump if less or equal/ not greater

unsigned

signed

```
        MOV CX, 5
next:   …
        …
        DEC CX      ;decrement CX
        JNZ next    ;if CX ≠ 0 (ZF = 0), jump to next
```

In the above example, the DEC instruction decrements the CX register by 1 and when CX becomes 0, it sets the zero flag (ZF) to 1, otherwise 0. Thus, the DEC (i.e., decrement instruction) updates the ZF and prepares the condition for the JNZ instruction. The JNZ instruction checks the status of the ZF and if it is 0, it jumps to the instruction with label `next`. When finally, CX becomes 0, the ZF is set to 1 which makes the condition false for the JNZ instruction and the control moves to the next instruction after JNZ.

The next four instructions in Table 1 are used while dealing with the unsigned numbers. They use the word "above" and "below." The 8-bit unsigned number 0101 0011 (decimal 83) is above the 8-bit unsigned number 0000 0101 (decimal 5). These instructions are normally used after the compare instruction. For example,

```
CMP AL, BL
JA next
```

The meaning of the above instructions together is "jump if AL is above BL," that is, if the 8-bit unsigned number in AL is above the 8-bit unsigned number in BL (unsigned number AL > unsigned number BL).

The last four instructions in Table 1 are very similar to previous four instructions except that they are used while dealing with the signed numbers. They use the words "greater" and "less." The 8-bit signed number 0000 0101 (decimal 5) is greater than the 8-bit signed number 1111 1100 (decimal –4). The meaning of

```
CMP AX, BX
JLE next
```

instructions is "jump if the signed number in AX is less than or equal to the signed number in BX (i.e., signed number AX <= signed number BX).

The important point to remember about the conditional jump instructions is that they are short jump instructions. This means that the range of jump for all the conditional jump instructions listed in Table 1 is –128 to +127 bytes from the address of the next instruction after the conditional jump instruction. Care should be taken while using these instructions in programs to ensure that the target instruction is within the range of –128 to +127 bytes from the address of the next instruction.

The conditional jump instructions are very useful for decision making and looping. We will see the use of the conditional jump instructions in implementing decision making and looping in programming examples in the next section.

Check Points

We now know that

- the conditional jump instruction performs jump only if the specified condition is true, otherwise it acts as NOP and the control moves to the next instruction.
- the 8086 conditional flags provide the conditions for the conditional jump instructions.
- the 8086 also provides the separate conditional instructions to deal with unsigned and signed numbers. They are used after the compare instruction.
- the 8086 conditional jump instructions are short jumps, that is, their range is –128 to +127 byte from the address of next instruction.

4.4 Decision Making and Looping

The data transfer, arithmetic and logical instructions are sequential in nature and they transfer the control to the next instruction after performing the specified operation. We can write only sequential programs using these instructions. The sequential programs are very simple in nature and cannot solve the complex real-life problems. The complex programs include not only sequence but also frequently use decision making and looping. Decision making allows the user to make the selection of the code to be executed based on the conditions, whereas looping provides the mechanism for repeating a particular block of code multiple times. To implement decision making and looping, we need the instructions which allow us to transfer the control arbitrarily to any instruction in the program. This is possible through the unconditional and conditional jump instructions. We have already studied them in the earlier sections. This section shows the use of them to implement decision making and looping with the help of the programming examples.

Let us first understand the implementation of decision making using the jump instructions. Decision making is same as the if-else statement in a high-level programming language and its general syntax is as follows:

```
If (Condition)
Then
        Statement block;
Else
        Statement block;
```

If the condition is true, then the statement block associated with the `If` part is executed, otherwise the statement block associated with the `Else` part is executed. Normally, the `Else` part is optional, and in case it is not present then the no action is performed if the condition is false. Figure 5 shows the implementation of if-else in 8086 using the status of the carry flag (CF) as condition.

As can be seen from Fig. 5(a), when the CF is 1, the condition is true and the statements in the statement block are executed. If the CF is 0, the condition is false and the control is transferred to the next instruction after the statement block. In Fig. 5(b), if the condition is true, that is CF = 1, the statement block-1 is executed, and if the condition is false, that is CF = 0, the statement block-2 is executed. After executing either of the statement blocks, the control is transferred to an instruction after the whole if-else statement. We can use any condition jump instruction to provide the condition in the above case which uses one or more conditional flags to provide the necessary conditions. The following programs show the use of decision making in programs.

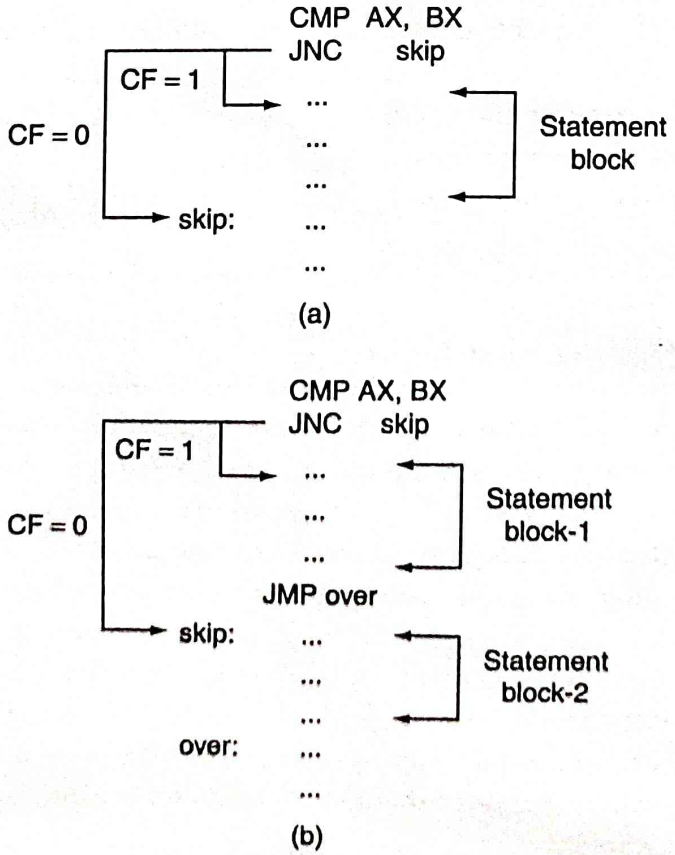


Figure 5 Implementing decision making in 8086 programming (a) without else and (b) with else.

Program 4.1

Write a program to find whether two 16-bit numbers are equal. Print the appropriate message.

Solution

The program is

```
data    SEGMENT
        n1    DW 1234h
        n2    DW 0A245h
        mes1 DB "Both the numbers are same $"
        mes2 DB "Both the numbers are different $"
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data                ;initialize
        MOV DS, AX

        MOV AX, n1                  ;get n1 in AX
        CMP AX, n2                  ;compare AX with n2

        MOV AH, 9                   ;print message
        JNZ noteq
        LEA DX, mes1                ;equal
        JMP over
noteq:  LEA DX, mes2                ;not equal
over:   INT 21h

        MOV AX, 4C00h               ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.1, both the numbers are compared and the message is printed based on the status of the zero flag (ZF). If both the numbers are equal then the compare instruction sets the ZF to 1, otherwise 0. In the above example, the message printed is "`Both the numbers are different`" as the numbers are not equal. You can test the program by giving the same value for both the numbers.

Program 4.2

Write a program to find the minimum of two 16-bit numbers.

Solution

The program is

```
data    SEGMENT
        n1 DW 1234h
```

```
            n2 DW 0A245h
            min DW ?
data        ENDS

code        SEGMENT
            ASSUME cs:code, ds:data

start:      MOV AX, data              ;initialize
            MOV DS, AX

            MOV AX, n1                ;get n1 in AX
            CMP AX, n2                ;compare AX with n2
            JC skip                   ;AX < n2
            MOV AX, n2                ;AX > n2, store n2 in AX
skip:       MOV min, AX               ;store AX to min

            MOV AX, 4C00h             ;terminate
            INT 21h
code        ENDS
            END start
```

As can be seen from Program 4.2, both the numbers are compared and the message is printed based on the status of the carry flag (CF). After comparison, if CF = 1, the n1 is small, otherwise n2 is small. Whichever number is smaller, it is kept in the AX register and finally stored into the min.

Program 4.3

Write a program to check whether the given 16-bit number is odd or even. Print the appropriate message.

Solution

The program is

```
data        SEGMENT
            num  DW 3D2Bh
            mes1 DB "Number is even $"
            mes2 DB "Number is odd $"
data        ENDS

code        SEGMENT
            ASSUME cs:code, ds:data

start:      MOV AX, data              ;initialize
            MOV DS, AX

            MOV AX, num               ;get num in AX
```

```
        AND AX, 0001h          ;AND with 1
        CMP AX, 0001h          ;check LSB

        MOV AH, 9              ;print message
        JZ skip
        LEA DX, mes1           ;odd
        JMP over
skip:   LEA DX, mes2           ;even
over:   INT 21h

        MOV AX, 4C00h          ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.3, the LSB of the numbers is checked to find whether the number is odd or even. If the LSB is 1, the number is odd, otherwise it is even. To check the LSB, the number is masked with `0001h` (only LSB = 1, rest of the bits are 0). If the number is odd, the result of the masking is `0001h`, otherwise it is `0000h`. Then the result is compared with the `0001h` and based on the status of zero flag (ZF), the message is printed. In the above case, the number is odd and hence the message printed is "`Number is odd`". You can test the program by giving the even value for the variable num. The above program can be easily implemented using the shift instructions. The LSB can be shifted into carry flag (CF), and based on the value of CF (either 0 or 1), the decision can be made. We will learn the shift instructions in Chapter 5.

The above examples implement the simple if-else structure provided in high-level programming languages. The high-level languages also provide the multiple if-else, also known as ladder if-else. This can be easily implemented in the 8086 programming by extending the concept used to implement simple if-else by using multiple conditions.

Looping is a technique to run the same block of instructions multiple times. In other words, looping allows us to perform the same operation many times. The general form of looping as provided by the most of the high-level languages is as follows:

```
Initialization;
While (Condition)
Begin
        Statement block;
        Reinitialization;
End
```

First, the initialization of looping variables is done. It is done only once. Then the condition involving looping variables is checked, and if is true, the body of the loop enclosed in the `Begin-End` is executed. The body of the loop contains the statement block followed by reinitialization. After performing the statement block, the reinitialization part updates the looping variables. The condition is checked

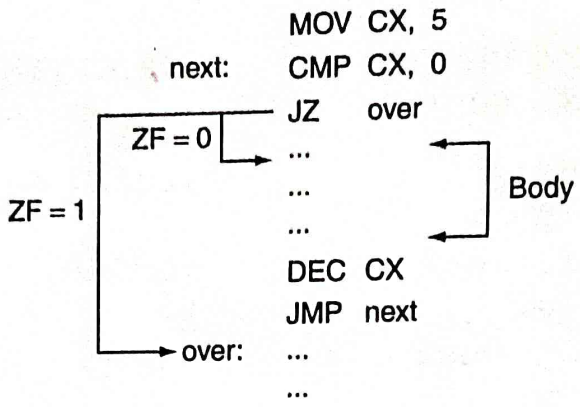


Figure 6 Implementing a loop in 8086 programming.

again with the updated value, and if it is true, the same process is repeated. It is repeated as long a the condition is true. When the condition is false, the loop is completed and the control is transferre to a statement after the `Begin-End` block. Figure 6 shows the implementation of looping in 808 programming.

As can be seen from Fig. 6, the CX register acts as looping variable which is initialized to an initia value 5. Then the CX is compared with 0, and if it is not 0, then the body is executed. The last instruc tion in the body is DEC CX which reduces the CX by 1 and then the control is transferred back to th compare instruction to check the condition again. If the condition is true, the process is repeated again. When the condition is false, the control is transferred to an instruction after the JMP instruction. In this case, the body of the loop is executed 5 times and hence CX acts as count indicating number of times the loop is to be executed. The loop in Fig. 6 is more conveniently written as follows:

```
        MOV CX, 5          ;initialize count
next:   ...                ;body
        ...
        ...
        DEC CX             ;reinitialization
        JNZ next           ;goto next, if ZF ≠ 1
```

The following programs demonstrate the use of looping in the 8086 programs.

Program 4.4

Write a program to find the sum of first N positive integers.

Solution

The program is

```
data    SEGMENT
        N    DW 000Ah
        sum DW ?
data    ENDS
```

```
code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data          ;initialize
        MOV DS, AX

        MOV CX, N             ;get N into CX
        MOV AX, 0             ;sum in AX = 0
        MOV BX, 1             ;First num in BX = 1
next:   ADD AX, BX            ;ADD next number
        INC BX                ;increment BX
        DEC CX                ;decrement N
        JNZ next              ;if not 0, goto next
        MOV sum, AX           ;store AX into sum

        MOV AX, 4C00h         ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.4, the last number N is stored in CX, the sum is stored in AX and hence it is initialized to 0, and the BX contains the next number to be added starting from 1. Each time in the loop, BX is added to partial sum in AX, BX is incremented by 1 to get the next number in sequence and CX is reduced by 1. If the CX does not reach to 0, the process is repeated, otherwise the control moves to the next instruction which stores the sum value in the AX register to the data segment in variable sum. As the value of N in the data segment is 000Ah (decimal 10), the answer would be 0037h (decimal 55). You can check that by executing program in the debug mode.

Program 4.5

Write a program to add given N 16-bit numbers.

Solution

The program is

```
data    SEGMENT
        block DW 0005h,5234h,4512h,1215h,213Dh,6D2Fh
              DW ?
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data          ;initialize
        MOV DS, AX
```

```
        MOV SI, OFFSET block      ;intialize pointer
        MOV CX, WORD PTR [SI]     ;get count in CX
        ADD SI, 2                 ;point to first number
        MOV AX, WORD PTR [SI]     ;get first num in AX
next:   ADD SI, 2                 ;point to next number
        DEC CX                    ;decrement CX
        JZ over                   ;if 0, goto over
        ADD AX, WORD PTR [SI]     ;add next number
skip:   JMP next                  ;goto next
over:   MOV WORD PTR [SI], AX     ;store result
        MOV AX, 4C00h             ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.5, the data segment defines an array `block` storing the size of th array followed by array elements. In our case, the size is 5 and then five 16-bit numbers are stored After array, the memory for storing a word is reserved to store the result. In the code segment, th pointer SI is set to point to `block` and the size of array is stored in the CX register. Then the firs number is stored in the AX register. Within the loop, the pointer is incremented to point to the nex element in the array and the CX register is decremented. If CX is 0, all the numbers are added, an the control moves out of the loop and stores the result in the next free word position. If the CX is no 0, the next number is added to the partial addition value in the AX, and the process is repeated. Thi program works for the array of size 1 or more. The result is always stored at the end of the array afte the last array element.

The complex real-life problems include both decision making and looping. The following program demonstrate the use of decision making and looping together to write more complex programs to solv real-life problems.

Program 4.6

Write a program to find the minimum from a block of N 16-bit numbers.

Solution

The program is

```
data    SEGMENT
        block DW 0005h,5234h,4512h,1215h,0D213h,0AD2Fh
              DW ?
data    ENDS
```

```
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data                ;initialize
        MOV DS, AX
        MOV SI, OFFSET block        ;intialize pointer
        MOV CX, WORD PTR [SI]       ;get count in CX
        ADD SI, 2                   ;point to first number
        MOV AX, WORD PTR [SI]       ;get first num in AX
next:   ADD SI, 2                   ;point to next number
        DEC CX                      ;decrement CX
        JZ over                     ;if 0, goto over
        CMP AX, WORD PTR [SI]       ;compare AX with next num
        JC skip                     ;AX is small
        MOV AX, WORD PTR [SI]       ;store next num in AX
skip:   JMP next                    ;goto next
over:   MOV WORD PTR [SI], AX       ;store smallest
        MOV AX, 4C00h               ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.6, the data segment defines an array `block` storing the size of the array followed by array elements. In our case, the size is 5 and then five 16-bit numbers are stored. After array, the memory for storing a word is reserved to store the smallest number. In the code segment, the pointer SI is set to point to `block` and the size of array is stored in the CX register. Then the first number is stored in the AX register. Within the loop, the pointer is incremented to point to the next element in array and the CX register is decremented. If the CX is 0, the comparisons are completed, and the control moves out of the loop and stores the answer in the next free word position. If the CX is not 0, the smaller value so far in AX is compared with the next value and the smaller between them is kept in the AX, and the process is repeated. This program even works for one number, as we check the counter initially in the loop after reducing it by 1. If there is only one number in the array, first time in the loop the CX after decrementing becomes 0 and the control comes out and stores the only number as smallest in the next free word position. Test the program for different array sizes and verify the answer by executing the program in the debug mode. You will get the result in the next word position after your array elements. For example, in the above case for five words, the result will be available after the fifth element in the data segment.

Program 4.7

Write a program to reverse a given array of 16-bit numbers.

Solution

The program is

```
data    SEGMENT
   block DW 0006h,1234h,5234h,4512h,1215h,0D213h,0AD2Fh
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data                    ;initialize
        MOV DS, AX

        LEA SI, block                   ;point to the size
        MOV CX, WORD PTR [SI]           ;get size into CX
        ADD SI, 2                       ;SI pointing to start
        MOV DX, CX                      ;multiply CX by 2
        MOV AL, DL
        MOV BL, 2
        MUL BL
        MOV DX, AX                      ;DX = CX x 2
        SUB DX, 2                       ;reduce DX by 2
        MOV DI, SI
        ADD DI, DX                      ;DI pointing to end
        MOV AX, CX                      ;divide CX by 2
        MOV BL, 2
        DIV BL
        MOV CX, 0
        MOV CL, AL
next:   MOV AX, WORD PTR [SI]           ;exchange the numbers
        MOV BX, WORD PTR [DI]           ;pointed by SI
        MOV WORD PTR [SI], BX           ;and DI pointers
        MOV WORD PTR [DI], AX
        ADD SI, 2                       ;SI = SI + 2
        SUB DI, 2                       ;DI = DI - 2
        DEC CX                          ;decrement CX
        JNZ next                        ;if not 0, goto next

        MOV AX, 4C00h                   ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.7, the data segment stores an array with name `block` with the first number as the size of the array followed by the elements of the array. In the code segment, the SI register after getting the size into CX, points to the first element of the array, and the DI register is initialized to point to the last element in the array by computing offset of last element. To compute the offset of the last element, 2 × (size of array − 1) is added to the start position, that is, SI. The total number of exchanges required is the size of the array divided by 2. The CX register contains the number of exchanges needed before the start of the loop. Then the loop is executed CX times. Each time in the loop, the elements at the offset SI and DI are exchanged, the SI is incremented by 2 to point to the next element, and the DI is decremented by 2 to point to the previous element. At the end of the loop execution, the whole array is reversed. Test the program by executing it in the debug mode and then verifying the contents in the data segment.

Program 4.8

Write a program to sort the given block of 8-bit numbers.

Solution

The program is

```
data    SEGMENT
        block DB 05h, 23h, 0A2h, 66h, 12h, 7Dh
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data            ;initialize
        MOV DS, AX

        LEA SI, block           ;point to size
        MOV CX, 0               ;get size
        MOV CL, BYTE PTR [SI]   ;into CL
        DEC CX                  ;set no. of passes
outer:  MOV DX, CX              ;set no. of comparisons
        INC SI                  ;set SI pointer
        MOV DI, SI              ;set DI = SI + 1
inner:  INC DI
        MOV AL, BYTE PTR [SI]   ;compare elements at
        CMP AL, BYTE PTR [DI]   ;position SI and DI
        JBE skip                ;if (SI) < (DI), goto skip
        MOV BL, AL              ;otherwise exchange them
        MOV AL, BYTE PTR [DI]
        MOV BYTE PTR [DI], BL
        MOV BYTE PTR [SI], AL
```

```
skip:   DEC DX                  ;decrement DX
        JNZ inner               ;if not 0, goto inner
        DEC CX                  ;decrement CX
        JNZ outer               ;if not 0, goto outer

        MOV AX, 4C00h           ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.8, the data segment stores an array with name `block` with the size of array as first number followed by elements. This program implements the sorting method in which for N elements, total of $N-1$ passes are performed, and in each pass the smallest value comes to the position equal to that pass number. This means that in the first pass the smallest value comes to the first position, in the second pass the second smallest value comes to the second position and so on. In each pass, SI points to the position equal to the pass and the value at that position is compared to the values at positions SI + 1 to the last position pointed by the DI register. At any time if the value pointed by SI is greater than the value pointed by DI, then they are exchanged. After completing all the passes, the values are arranged in ascending order. In the above case, $N = 5$ and hence four passes are needed to sort the array. Testing the above program by executing it in the debug mode, the array elements will look as follows:

```
12h 23h 66h 7Dh A2h
```

Program 4.9

Write a program to find factorial of a given 8-bit number. Give the number such that its factorial is less than or equal to FFFFh.

Solution

The program is

```
data    SEGMENT
        N     DB     05h
        fact DW ?
data    ENDS

code    SEGMENT
        ASSUME cs:code , ds:data
start:  MOV AX, data            ;initialize
        MOV DS, AX

        MOV CX, 0               ;set CH = 0
        MOV CL, N               ;get N into CL
```

```
        MOV AX, 1               ;set AX = 1
        MOV BX, 1               ;set BX = 1
next:   MUL BX                  ;AX = AX x BX
        INC BX                  ;increment BL
        CMP BX, CX              ;is BX <= N?
        JBE next                ;if yes, goto next
        MOV fact, AX            ;store result

        MOV AX, 4C00h           ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.9, the 8-bit number N whose factorial is to be found is stored in the CX register. The AX and BX registers are initialized to 1. Within the loop, the AX and BX are multiplied. The result of the multiplication is stored in the DX:AX register pair, but the DX remains zero and hence our partial multiplication is only in the AX part. Then the BX is incremented and compared with the CX register, that is, N. If it is less than or equal to N, the control is transferred to the beginning of the loop. Otherwise, the loop is completed and the $N!$ (i.e., $1 \times 2 \times 3 \times \cdots \times N$) is computed and stored in the data segment as variable `fact`.

Check Points

We now know that

- decision making and looping are important techniques to write complex programs.
- decision making is similar to the if-else statement in high-level languages and allows the user to select the block of statements for execution.
- looping is similar to the while-do loop in high-level languages and allows the user to execute block of codes multiple times.
- real-life problems involve different combinations of decisions and repetitions and hence decision-making and looping techniques are important to implement the programs for solving them.

4.5 Loop Instructions

The looping instructions, also known as *iteration control instructions*, are used to control the number of iterations to be performed on a specified block of instructions forming the body of a loop. These include the LOOP, LOOPE/LOOPZ and LOOPNE/LOOPNZ instructions as well as special instruction JCXZ to skip the block of instructions when CX is zero. Let us understand these instructions through their use in examples.

LOOP Instruction

It is the instruction which performs looping until CX = 0. The syntax of the LOOP instruction is as follows:

```
LOOP target_inst
CX ← CX - 1
;Jump to instruction labeled with target_inst if CX ≠ 0
```

The LOOP instruction repeats the block of instructions CX number of times. The CX register is used as count register. When the LOOP instruction is executed, the CX register is automatically decremented by 1, and if the CX is not 0 then the control is transferred to the instruction whose address is represented by the label *target_inst*. If the CX is 0 then the control moves to the next instruction after the LOOP instruction. The displacement of the target instruction must be in the range of −128 to +127 bytes from the address of the instruction after the LOOP instruction. The LOOP instruction does not affect the flags. The function of the LOOP instruction can also be represented as follows:

```
CX ← CX - 1
if(CX ≠ 0) then
      IP ← IP + displacement
if(CX = 0) then
      move to the next instruction
```

Consider the following example:

```
        MOV CX, 5          ;load count in CX reg
next:   ...
        ...
        ...
        LOOP next          ;if CX ≠ 0, goto next
```

The MOV instruction loads the count in the CX register. The value loaded in the CX register denotes the iteration value as the block of instructions starting from the instruction labeled with `next` to the LOOP instruction is executed that many times. The LOOP instruction decrements the CX by 1 every time and sends the execution control to the target instruction with label `next` as long as CX ≠ 0. Finally, when CX = 0, it sends the control to the instruction next to the LOOP, that is, the control comes out of the loop.

The LOOP instruction provides more efficient way for performing the iterations. In the above example, if the LOOP instruction is not provided then we need to use DEC and JNZ instructions as follows:

```
        MOV CX, 5          ;load count in CX reg
next:   ...
        ...
        ...
        DEC CX
        JNZ next           ;if ZF ≠ 1, goto next
```

As can be seen from the above code snippet, for each iteration two instructions are executed, that is, DEC and JNZ, instead of one instruction, that is, LOOP.

Program 4.10

Write a program to generate the first 10 Fibonacci numbers starting from 0 and 1.

Solution

The program is

```
data    SEGMENT
        fib DB 10 dup(0)
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data            ;intialize
        MOV DS, AX

        MOV CX, 10              ;set count
        MOV SI, OFFSET fib      ;set pointer
        MOV AL, 0               ;Fisrt number
        MOV BYTE PTR [SI], AL   ;store
        INC SI                  ;increment pointer
        MOV AL, 1               ;second number
        MOV BYTE PTR [SI], AL   ;store
        INC SI                  ;increment pointer
        SUB CX, 2               ;reduce counter
        MOV AL, 0               ;intialize
        MOV BL, 1
next:   ADD AL, BL              ;get next number
        MOV DL, AL
        MOV BYTE PTR [SI], AL   ;store result
        INC SI                  ;increment pointer
        MOV AL, BL              ;reinitialize
        MOV BL, DL
        LOOP next               ;goto next, if count is not 0

        MOV AX, 4C00h           ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.10, the first two numbers, 0 and 1, are stored in the array fib. In the loop, every time the last two numbers in AL and BL registers are added, the result is stored in array fib at the next position and AL and BL are reinitialized. If you see the output in debug, it looks as follows:

```
00h   01h   01h   02h   03h   05h   08h   0Dh   15h   22h
```

LOOPE/LOOPZ Instruction

This is known as loop if equal and performs looping while CX ≠ 0 and ZF = 1. The syntax of the LOOPE/LOOPZ instruction is as follows:

```
LOOPE/LOOPZ target_inst
CX ← CX - 1
;Jump to instruction labeled with target_inst if CX ≠ 0
;and ZF = 1
```

The LOOPE and LOOPZ are two different mnemonics for the same instruction. The count representing the iteration value is loaded into the CX register. This instruction loops while CX ≠ 0 and ZF = 1. This means that it loops CX number of times or until ZF = 0. When it executes, the CX register is automatically decremented by 1, and if CX ≠ 0 and ZF = 1 then the control is transferred to the instruction whose address is represented by the label `target_inst`. If CX = 0 or ZF = 0 then the control moves to the next instruction after the LOOPE/LOOPZ instruction. The displacement of the target instruction must be in the range of – 128 to + 127 bytes from the address of the instruction after the instruction LOOPE/LOOPZ. This instruction does not affect the flags. The function of LOOPE/LOOPZ instruction can also be represented as follows:

```
CX ← CX - 1
if(CX ≠ 0 and ZF = 1) then
     IP ← IP + displacement
if(CX = 0 or ZF = 0) then
     move to the next instruction
```

The LOOPE/LOOPZ instruction is normally used after the compare instruction. If comparison is equal (ZF = 1) and the CX ≠ 0 then the looping is performed. Let us see the use of the LOOPE/LOOPZ instruction in the following program to check whether all the numbers in an array are same or not.

Program 4.11

Write a program to check whether all the numbers in a given array of 16-bit numbers are same. Print the appropriate message.

Solution

The program is

```
data    SEGMENT
        array DW 1234h, 1234h, 1234h, 1234h, 1234h
        mes1  DB "All the numbers are same $"
        mes2  DB "All the numbers are not same $"
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data
```

```
start:   MOV AX, data                ;intialize
         MOV DS, AX

         MOV CX, 5                   ;set count
         MOV SI, OFFSET array        ;set pointer
         MOV AX, WORD PTR [SI]       ;get first no. in AX
         DEC CX                      ;decrement count
next:    ADD SI, 2                   ;point to next number
         CMP AX, WORD PTR [SI]       ;compare with AX
         LOOPE next                  ;repeat if equal

         MOV AH, 9                   ;print message
         JNZ skip                    ;Not same
         LEA DX, mes1                ;all are same
         JMP print
skip:    LEA DX, mes2
print:   INT 21h

         MOV AX, 4C00h               ;terminate
         INT 21h
code     ENDS
         END start
```

As can be seen from Program 4.11, the first number from the array is moved into the AX register and then the rest of the numbers are compared with the number in AX. The control comes out of the loop either when comparison fails, that is, ZF ≠ 1 or CX = 0. Then using the status of the ZF, the appropriate message is printed. In the above case, the message "`All the numbers are same`" is printed as all the numbers in the array `array` are same. If you test the program by changing one of the numbers in the array `array` in the above program, the message "`All the numbers are not same`" gets printed.

LOOPNE/LOOPNZ Instruction

This is known as loop if not equal and performs looping while CX ≠ 0 and ZF = 0. The syntax of the LOOPNE/LOOPNZ instruction is as follows:

```
LOOPNE/LOOPNZ target_inst
CX ← CX - 1
;Jump to instruction labeled with target_inst if CX ≠ 0
;and ZF = 0
```

The LOOPNE and LOOPNZ are two different mnemonics for the same instruction. The count representing the iteration value is loaded into the CX register. This instruction loops while CX ≠ 0 and ZF = 0. This means that it loops CX number of times or until ZF = 1. When it executes, the CX register is automatically decremented by 1, and if CX ≠ 0 and ZF = 0 then the control is transferred to the instruction whose address is represented by the label *target_inst*. If CX = 0 or ZF = 1 then

the control moves to the next instruction after the LOOPNE/LOOPNZ instruction. The displacement of the target instruction must be in the range of −128 to +127 bytes from the address of the instruction after the instruction LOOPNE/LOOPNZ. This instruction does not affect the flags. The function of LOOPNE/LOOPNZ instruction can also be represented as follows:

```
CX ← CX - 1
if(CX ≠ 0 and ZF = 0) then
        IP ← IP + displacement
if(CX = 0 or ZF = 1) then
        move to the next instruction
```

The LOOPNE/LOOPNZ instruction is normally used after the compare instruction. If comparison is not equal (ZF = 0) and the CX ≠ 0 then the looping is performed.

JCXZ Instruction

This is known as jump if CX = 0. The syntax of the JCXZ instruction is as follows:

```
JCXZ target_inst
;jump to instruction labeled with target_inst if CX = 0
```

The JCXZ instruction transfers the control to a target instruction if CX = 0 when it executes. It is normally used to skip the block of instructions when CX = 0. The displacement of the target instruction must be in the range of −128 to +127 bytes from the address of the instruction after the instruction JCXZ. This instruction does not affect the flags. The function of JCXZ instruction can also be represented as follows:

```
if(CX = 0) then
    IP ← IP + displacement
else
    move to the next instruction
```

Consider the following code snippet:

```
        MOV CX, 5          ;set the loop count
next:   JCXZ over          ;complete the loop if CX = 0
        …                  ;body of the loop
        …
        DEC CX             ;CX = CX - 1
        JMP next           ;goto next
over:   …                  ;out of loop
```

As can be seen in above example, the move instruction initializes the CX with 5 as loop count. This means that the loop performs five iterations. In the beginning of the loop, the JCXZ instruction checks the value of the CX register. At the end, every time in the loop the CX is reduced by 1 and control is transferred using the jump instruction to the beginning of the loop. After five iterations, the value of CX becomes 0 and the JCXZ instruction sends the control to an instruction with label `over` and comes out of the loop.

Check Points

We now know that

- the 8086 provides special iteration control instructions also known as looping instructions for efficient looping.
- these instructions include LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ and JCXZ instructions.
- the LOOP instruction repeats the block of instructions while CX ≠ 0.
- the LOOPE/LOOPZ instruction performs looping while CX ≠ 0 and ZF = 1. It is used normally after the compare instruction.
- the LOOPNE/LOOPNZ instruction performs looping while CX ≠ 0 and ZF = 0. It is used normally after the compare instruction.
- the JCXZ instruction is used to skip the block of instructions when CX = 0.

4.6 ASCII and BCD Arithmetic

The processors normally deal with the binary numbers. However, many processors allow the processing of numbers in other formats like ASCII and BCD. The ASCII is a 7-bit code used to represent digits, alphabets and special characters. The BCD code is used to represent the decimal numbers using 4-bit binary code for each decimal digit 0–9. The 8086 also provides instructions to support the ASCII and BCD numbers within the arithmetic group of instructions. These include AAA (ASCII Adjust for Addition), DAA (Decimal Adjust for Addition), AAS (ASCII Adjust for Subtraction), DAS (Decimal Adjust for Subtraction), AAM (ASCII Adjust for Multiplication) and AAD (ASCII Adjust for Division). Let us understand all these instructions with the help of examples.

AAA Instruction

This instruction is known as "ASCII Adjust for Addition." The syntax for AAA instruction is as follows:

```
AAA
;AL adjusted to ASCII for addition
```

When we enter the digits in the computer using keyboard, the ASCII values of digits are transmitted. The ASCII values of digits 0–9 are 30h to 39h in sequence. If we add two ASCII values of two decimal digits using the ADD instruction, they are added as if they are binary numbers, and the result is not in the desired format. The AAA instruction is used to convert such result of addition into unpacked BCD. The AAA instruction is used after the addition instruction and works only on the AL register. The AAA instruction updates the AF and CF flags, while other flags remain undefined. Let us understand its working with help of the following example. The addition of 5 and 7 is 12. Assume that AL contains the ASCII of 5 (35h) and BL contains ASCII of 7 (37h).

```
;AL = 0011 0101,  BL = 0011 0111 before addition
ADD AL, BL      ;Answer in AL = 0110 1100 = 6Ch
AAA             ;AL = 0000 0010 unpacked BCD 2
                ;CF = 1
                ;Hence, Answer is 12 decimal
```

The answer of addition is not in the correct BCD form. The AAA instruction converts the AL in unpacked BCD 2 with all the bits in AH zero and makes CF = 1. Combining the CF and AL, th result is decimal 12. If we want the result in ASCII, we can OR the contents of AL after AAA with 30 In this case, the ORing of unpacked BCD 2 with 30h(0000 0010 | 0110 0000 = 0110 0010) resul into 32h which is the ASCII value of digit 2.

DAA Instruction

This instruction is known as "Decimal Adjust for Addition" and converts AL to BCD after the additio of two packed BCD numbers. The syntax of the DAA instruction is as follows:

```
DAA
;Adjust AL after the BCD addition
```

The addition of two packed BCD numbers produced by the ADD instruction is not in the BCD form The result stored in the AL register after addition of two packed BCD numbers is converted to BCI by the DAA instruction. To convert the contents of AL to BCD, the DAA instruction adds 6 to th lower nibble of the AL register if it is greater than 9 or the AF flag was set to 1 by the addition. Now the upper nibble of AL is greater than 9 or the CF was set to 1 by addition or correction, then the DA instruction adds 6 to the upper nibble of the AL register. The DAA instruction affects AF, PF, CF, Z flags and OF remains undefined after the DAA instruction.

The DAA instruction is used to perform the decimal addition. As we know, the decimal number are stored using BCD codes. If both the nibbles of the 8-bit register contain BCD code then it i called *packed BCD number*. For example, 0110 0010 represents the number 62 in BCD. The addition instruction adds two BCD numbers as if they are some binary numbers and hence the result is no correct BCD number. The DAA instruction is used to correct the result of adding two packed BCI numbers to the correct BCD number. The following example clears the idea.

```
;AL = 0110 0101 = 65 BCD, BL = 0010 0111 = 27 BCD
ADD AL, BL      ;Result AL = 1000 1100 = 8Ch, AF=CF=0
DAA             ;AL = 1001 0010 = 92 BCD
```

The result of addition in the above example is not in the correct BCD form. The DAA instruction adds 0110, that is 6, to AL as the lower nibble of AL (1100) is greater than 9 resulting into 1001 0010 (92 BCD). Now the upper nibble is not greater than 9 and CF is set by neither addition nor correction; hence the second step to correct the upper nibble is not needed. Consider another example as follows:

```
;AL = 0110 1001 = 69 BCD, BL = 0100 0111 = 47 BCD
ADD AL, BL      ;Result AL = 1011 0000 = B0h, AF=1,CF=0
DAA             ;AL = 0001 0110 = 16 BCD, CF = 1
```

In the above example, the result of addition is AL = B0h. The lower nibble is not greater than 9 but the AF = 1 and hence DAA adds 0110 to the lower nibble resulting into B6h. Now the upper nibble is greater than 9, so the DAA adds 0110 to the upper nibble of B6h resulting into 16 into the AL register with CF = 1. Combining the CF and AL, the result is 116 (69 + 47).

Program 4.12

Write a program to generate first 10 Fibonacci numbers in decimal form starting from 0 and 1.

Solution

The program is

```
data    SEGMENT
        fib DB 10 dup(0)
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data            ;intialize
        MOV DS, AX

        MOV CX, 10              ;set count
        MOV SI, OFFSET fib      ;set pointer
        MOV AL, 0               ;Fisrt number
        DAA                     ;convert to BCD
        MOV BYTE PTR [SI], AL ;store
        INC SI                  ;increment pointer
        MOV AL, 1               ;second number
        DAA                     ;convert to BCD
        MOV BYTE PTR [SI], AL ;store
        INC SI                  ;increment pointer
        SUB CX, 2               ;reduce counter
        MOV AL, 0               ;intialize
        MOV BL, 1
next:   ADD AL, BL              ;get next number
        DAA                     ;convert to BCD
        MOV DL, AL
        MOV BYTE PTR [SI], AL ;store result
        INC SI                  ;increment pointer
        MOV AL, BL              ;reinitialize
        MOV BL, DL
        LOOP next               ;goto next, if count is not 0

        MOV AX, 4C00h           ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.12, the initial two Fibonacci numbers are converted to BCD and stored in the data segment into the array fib. Then in a loop, the addition of the last two Fibonacci numbers

in the BCD form stored in AL and BL registers is done, and the result is converted into BCD and stored as the next number in array fib. At the end of the loop, the AL and BL registers are reinitialized with the last two Fibonacci numbers. The loop is executed eight times to generate the next eight Fibonacci numbers. After assembling and linking the above program, if you debug it and see the data segment, you will find the numbers as follows:

```
00    01    01    02    03    05    08    13    21    34
```

AAS Instruction

This instruction is known as "ASCII Adjust for Subtraction." The syntax for AAS instruction is as follows:

```
AAS
;AL adjusted to ASCII for subtraction
```

The AAS instruction works in the same manner as the AAA instruction. After subtraction of one ASCII value from another ASCII value of a digit, we can use the AAS instruction to adjust the answer in the AL register to unpacked BCD form. It stores the unpacked BCD in the lower nibble of the AL register and clears the upper nibble of the AL register. Note that the AAS instruction works only on the AL register. It updates only the AF and CF flags while other flags remain undefined. The following examples demonstrate the working of the AAS instruction.

Assume that we want to subtract the ASCII of digit 3 from the ASCII of digit 8. This requires initializing AL with 0011 1000 = 38h = ASCII of 8 and BL with 0011 0011 = 33h = ASCII of 3. The following instructions produce the unpacked BCD 5 as answer in the AL register:

```
SUB AL, BL      ;Result AL = 0000 0101 = 5, CF = 0
AAS             ;AL = 0000 0101 = unpacked BCD 5,CF = 0
```

Consider another example to perform ASCII 3 – ASCII 8. This needs AL = 0011 0011 = 33h = ASCII of 3 and BL = 0011 1000 = 38h = ASCII of 8.

```
SUB AL, BL      ;Result AL = 1111 1011 = -5, CF = 1
AAS             ;AL = 0000 0101 = unpacked BCD 5
                ;CF = 1 as borrow
```

In the above case, the answer is –5 after subtraction as a large value is subtracted from a small value. The borrow value 1 in CF indicates that. The AAS instruction adjusts it to unpacked BCD 5 with borrow 1.

DAS Instruction

This instruction is known as "Decimal Adjust for Subtraction." It converts AL to BCD after the subtraction of two packed BCD numbers. The syntax of the DAS instruction is as follows:

```
DAS
;Adjust AL after the BCD subtraction
```

The subtraction of two packed BCD numbers produced by SUB instruction is not in the BCD form. The result stored in the AL register after subtraction of two packed BCD numbers is converted to BCD by the DAS instruction. To convert the contents of AL to BCD, the DAS instruction subtracts 6 from the lower nibble of the AL register if it is greater than 9 or if the AF flag was set to 1 by the subtraction. Now, if the upper nibble of AL is greater than 9 or the CF flag was set to 1, then the DAS instruction subtracts 6 from the upper nibble of the AL register. The DAS instruction affects the AF, PF, CF, ZF flags and OF remains undefined after the DAS instruction. The following example subtracts 47 from 75 and produces the result 28.

```
;AL = 0111 0101 = 75 BCD, BL = 0100 0111 = 47 BCD
SUB AL, BL      ;Result AL = 0010 1110 = 2Eh, AF=1,CF=0
DAS             ;AL = 0010 1000 = 28 BCD
```

AAM Instruction

This instruction is known as "ASCII Adjust for Multiplication." The syntax of the AAM instruction is as follows:

```
AAM
;Adjust AL to two unpacked BCD numbers in AX (AH:AL)
```

The AAM instruction adjusts the result stored into AL after multiplication of two unpacked BCD numbers to two unpacked BCD bytes into the AX register. The operand for the AAM instruction is always the AL register, and it affects the PF, SF and ZF flags while the other conditional flags remain undefined. The following example demonstrates the functioning of the AAM instruction:

```
;AL = 0000 0101 = unpacked BCD 5
;BL = 0000 0111 = unpacked BCD 7
MUL BL          ;AX = AL X BL = 0023h
AAM             ;AX = 0000 0011 0000 0101 = 0305
                ;AH = 0000 0011 = unpacked BCD 3
                ;AL = 0000 0101 = unpacked BCD 5
```

The above example performs the multiplication 5 × 7 and produces 35. The source operands are unpacked BCD numbers which are multiplied by the MUL instruction and then the result is converted to two unpacked BCD bytes into the AX register. Thus, the answer 35 is held by the AX register as two unpacked BCD bytes.

We can use the AAM instruction also with ASCII numbers. If source operands are ASCII numbers then before multiplication, we have to mask the 3 in the upper nibble of each source operand to convert them to unpacked BCD values. After doing this processing, the above instructions are performed to get the answer in two unpacked BCD bytes. To convert the answer into the ASCII form, we have to insert the 3 in upper nibble of both the unpacked BCD bytes stored into the AH and AL registers, respectively. The following sequence of instructions performs it:

```
;AL = 0011 0101 = 35h = ASCII 5
;BL = 0011 0111 = 37h = ASCII 7
AND AL, 0Fh       ;AL = 0000 0101 = unpacked BCD 5
```

```
AND BL, 0Fh        ;BL = 0000 0111 = unpacked BCD 7
MUL BL             ;AX = AL X BL = 0023h
AAM                ;AX = 0000 0011 0000 0101 = 0305
                   ;AH = 0000 0011 = unpacked BCD 3
                   ;AL = 0000 0101 = unpacked BCD 5
OR AX, 3030h       ;AH = 0011 0011 = 33h = ASCII 3
                   ;AL = 0011 0101 = 35h = ASCII 5
```

AAD Instruction

This instruction is known as "ASCII Adjust for Division." The syntax of the AAD instruction is as follows:

```
AAD
;Convert unpacked BCD bytes into AH and AL to binary number
;in AL before ;Division
```

The AAD instruction converts the two unpacked BCD digits stored in the AH and AL registers to an equivalent binary number which then is divided by the unpacked BCD byte. The division produces the quotients and remainders in the AL and AH registers in unpacked BCD form. The AAD instruction affects the PF, SF and ZF flags while the other unconditional flags remain undefined. The following example shows the working of the AAD instruction:

```
;AX = 0405h = unpacked BCD of decimal 45
;BL = 0000 0111 = unpacked BCD 7
AAD          ;AL = 0010 1101 = 2Dh = 45 decimal
DIV BL       ;Quotient = AL = 0000 0110 = unpacked BCD 6
             ;Remainder = AH = 0000 0011 = unpacked BCD 3
```

The AAD instruction can also be used with ASCII numbers like the AAM instruction with masking and ORing.

Check Points

We now know that

- the 8086 supports instructions that are used to perform the processing of numbers in ASCII and BCD formats.
- the AAA instruction converts the AL register to unpacked BCD after addition of two ASCII numbers.
- the DAA instruction converts the AL register to packed BCD form after addition of two packed BCD numbers.
- the AAS instruction converts the AL register to unpacked BCD after subtraction of two ACSII numbers.
- the DAS instruction converts the AL register to packed BCD form after subtraction of packed BCD numbers.
- the AAM instruction converts the AX register to two unpacked BCD bytes after multiplication of two unpacked BCD bytes.
- the AAD instruction converts the two unpacked BCD bytes in the AX register to a binary number in the AL register before division by an unpacked BCD byte.

4.7 Processor Control Instructions

The processor control instructions change the way the processor operates. The flag set/rest instructions, no operation instruction and external hardware synchronization instructions are included in the processor control instructions. Let us first discuss the flag set/reset instructions.

Flag Set/Reset Instructions

The flag set/reset instructions include clear carry flag (CLC), set carry flag (STC), complement carry flag (CMC), clear direction flag (CLD), set direction flag (STD), clear interrupt flag (CLI) and set interrupt flag (STI). We will discuss only instructions related to carry flag in this section as the instructions with regard to direction flags are covered in Chapter 5 and the instructions with regard to interrupt flags are covered in Chapter 7.

CLC Instruction

The syntax for clear carry flag (CLC) instruction is as follows:

```
CLC
;CF ← 0, clear the carry flag
```

This instruction clears the carry flag.

STC Instruction

The syntax for the set carry flag (STC) instruction is as follows:

```
STC
;CF ← 1, set the carry flag
```

This instruction sets the carry flag.

The carry flag is used to communicate the status of the execution of the procedure call or system call to the calling program. We have used the DOS system calls using INT 21h in Chapter 3. The procedures are discussed in Chapter 6. If the execution of the procedure or system call is successful, then the call is returned with clearing the carry flag, otherwise the control is returned with carry flag set to 1. The calling program can check the carry flag to determine the success of the call. The following example clarifies the concept:

```
        ...
        CALL proc
        JC fail
        ;call is successful
        ...
fail:   ;call is unsuccessful
        ...
```

As can be seen from the above example, after a call to a procedure named as `proc`, the carry flag is checked using the JC instruction. If the carry flag is 0, then the call is successful and the control goes to the next instruction after the JC instruction. If the carry flag is 1, then the call is unsuccessful and the control goes to the instruction labeled with label, "`fail`".

CMC Instruction

The syntax of the complement carry flag (CMC) instruction is as follows:

```
CMC
;~(CF), complement carry flag
if (CF = 0)
then
     CF ← 1
else
     CF ← 0
```

This instruction complements the carry flag, that is, if it is 0, it is made 1, and if it is 1, it is made 0.

The CLD and STD instructions are discussed in Chapter 5 while discussing the string instruction The CLI and STI instructions are discussed in Chapter 7 while covering the interrupts in detail.

Set/Reset TRAP Flag

Observe that the 8086 does not provide instructions for clearing and setting the trap flag. If the tr flag is set and the instruction is executed, the program stops after execution of the instruction a allows the user to see the register and memory contents. The trap flag is not used directly by the us but it is used normally by debugging tools to facilitate single-step execution. The trace "t" command DOS "debug" is the example of single-step. The debugging tool sets the trap flag before execution each instruction to allow single-step execution. As there is no instruction to set the trap flag, it is set 1 using the following instructions:

```
PUSHF                    ;push the flag on to stack
MOV BP, SP               ;store SP into BP
OR WORD PTR [BP+0], 0100h ;set the trap flag
POPF                     ;restore the trap flag
```

The first instruction copies the contents of the flag register onto the top of the stack. The second instruc tion copies the SP into BP so that using BP we can directly access the contents of the stack. Remembe that the trap flag is the lower-most bit in the higher byte of the flag register. The third instruction se the trap flag without changing other flags. There is no addressing mode which allows using alone BP a pointer, so it is used with displacement 0. Finally, the last instruction copies the contents of the top o the stack into the flag register. Hence, after executing the above sequence of instructions, the trap fla is set without changing the other flags. Similarly to reset the trap flag to 0, replace the third instructio in the above sequence with the following instruction:

```
AND WORD PTR [BP+0], 0FEFFh ;reset the trap flag
```

No Operation (NOP) Instruction

This is known as no operation instruction as it does nothing except killing the clock cycles. The syntax for the NOP instruction is as follows:

```
NOP
;no operation, waits for 3 clock cycles
```

This instruction simply waits for three clock cycles and then moves to the next instruction. It is normally used to delay the operation or is used to create the delay of a specific period using the delay loop or delay routine. The following is an example of the delay loop:

```
        MOV CX, 5
next:   NOP
        LOOP next
```

The above delay loop creates the delay equivalent to clock cycles needed by the move instruction plus the five times the clock cycles needed by the no operation and loop instructions.

External Hardware Synchronization Instructions

These instructions include the halt (HLT), escape (ESC), wait (WAIT) and lock (LOCK) instructions. They are briefly described in this section as their details are beyond the scope of this book.

HLT Instruction

The halt instruction halts the processing and enters into the halt mode. The syntax of the halt instruction is as follows:

```
HLT
;halt the processing
```

It actually performs the no operation until the processor comes out of the halt mode. The processor comes out of the halt mode either if the valid interrupt comes on INTR or NMI pins or it gets the reset signal on the reset pin.

ESC Instruction

The escape (ESC) instruction is used to transfer an instruction to the coprocessor such as 8087. The syntax of the ESC instruction is as follows:

```
ESC
;transfer an instruction to coprocessor
```

The 8087 is a math coprocessor used with 8086 to execute the floating point instructions in an efficient manner.

WAIT Instruction

The WAIT instruction is used to synchronize with the external coprocessor. The syntax of the WAIT instruction is as follows:

```
WAIT
;enter into idle or wait state
```

Once the WAIT instruction is executed, the 8086 enters into the idle mode, also called the wait mode. It comes out of the wait mode either if it gets a valid interrupt on INTR or NMI pin or gets the signal on the test pin. This instruction is normally used while switching from a coprocessor instruction to a processor instruction to synchronize the 8086 processor with the math coprocessor 8087.

LOCK Instruction

The LOCK instruction is used to prevent the other processor to take the control of shared resources like buses in multiprocessor systems. The syntax of the LOCK instruction is as follows:

```
LOCK
;prevent other processor to take control of buses
```

The lock instruction is used as prefix to another instruction to prevent the other processor to take control of shared resources in between the execution of the instruction that follows the LOCK. For example, the following instruction

```
LOCK IN AL, 81h
```

locks the system buses while reading the data from the device connected onto the port 81h.

Check Points

We now know that

- the 8086 provides the processor control instructions to control the way the processor operates.
- the processor control instructions include the flag set/reset instructions, the no operation instruction and the external hardware synchronization instructions.
- the flag set/rest instructions include CLC, STC, CMC, CLD, STD, CLI and STI instructions.
- there are no instructions to set or reset the trap flag.
- the no operation instruction (NOP) is used to create delays.
- the external hardware synchronization instructions are used to synchronize the 8086 with the external coprocessor like 8087. They include HLT, ESC, WAIT and LOCK instructions.

Summary

- The 8086 processor provides branch control instructions, which include unconditional and conditional jump instructions, and iteration control instructions to implement decision making and looping.
- The unconditional jump instructions provide the control transfer to the target instruction unconditionally.
- The unconditional jump instruction can be short or near and provide jump within the segment. It is also known as intrasegment jump as both the jump instruction and the target instruction are in the same segment.
- The range of the short jump is −128 to +127 bytes and that of the near jump is −32768 to +32767 bytes from the IP of the next instruction.
- The unconditional jump, which provides jump between two segments, is called the intersegment jump. It is useful to manage the complex code in modular fashion.
- The 8086 provides the conditional jump instructions that facilitate the branch control conditionally. The conditions for these instructions are provided by the status of conditional flags. All the conditional jump instructions are by default short jumps, that is, their range is −128 to +127 bytes.
- The iteration control instructions are provided by the 8086 to implement the looping efficiently. These include the LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ and JCXZ instructions. These instructions use the CX as the count register to control the number of iterations.

- The 8086 processor supports the ASCII and BCD arithmetic in addition to binary arithmetic using its special instructions like AAA, DAA, AAS, DAS, AAM and AAD.
- The processor control instructions include the flag set/reset instructions, no operation instructions and external hardware synchronization instructions.
- The flag set/reset instructions are available for carry flag, direction flag and interrupt flag. There is no instruction for the trap flag. The trap flag is set/reset by writing the sequence of instructions.
- The no operation instructions provide the time delay and are useful for implementing delay loops or delay subroutines.
- The external hardware synchronization instructions are used to synchronize the 8086 with external hardware devices. These include the HLT, ESC, LOCK and WAIT instructions.

Glossary

Unconditional jump is used to perform branch control unconditionally.

Forward jump is a jump in which the target instruction comes after the jump instruction.

Backward jump is a jump in which the target instruction comes before the jump instruction.

Near jump is an unconditional jump with the range −32768 to +32767 bytes from the IP of the next instruction.

Short jump is a jump with the range −128 to +127 bytes from the IP of the next instruction.

Intrasegment jump means a jump within a segment.

Intersegment jump means a jump between two segments.

Conditional jump means that a branch control is performed only if the specified condition is true.

Decision making means selection among the two or more alternatives using conditions.

Looping means to execute a group of instructions repeatedly.

Interaction control instructions are instructions used to perform iteration control, that is looping, efficiently.

ASCII stands for American Standard Code for Information Interchange.

BCD stands for Binary Coded Decimal in which each decimal digit is given a 4-bit binary code.

Unpacked BCD is an 8-bit number having only the lower nibble containing a BCD code whose upper nibble is always 0000.

Packed BCD is an 8-bit number with both the nibbles used to store the BCD code.

Processor control instructions are instructions which are used to set/reset the flags and for no operation and external hardware synchronization.

Objective Questions

State whether true/false. Give reason for your answer

1. The short jump can be used to jump within the whole segment.
2. The intersegment jump needs to change both CS and IP values.
3. The conditional jump instructions are always short jump instructions.
4. The LOOP instruction is always used after the compare instruction.
5. The DAA instruction works with unpacked BCD numbers.
6. The AAA instruction works only with the AL register.

7. The AAD instruction works after the division operation is performed.
8. The processor comes out of the halt mode using either the interrupt or the reset signal.
9. The NOP instruction is only a time delay and does not perform any operation.
10. The trap flag can be set by using the STI instruction.
11. The WAIT instruction is used to put the 8086 into idle mode.
12. The LOCK instruction is always used as prefix to another instruction.

Multiple-Choice Questions

1. The near jump instruction uses _____bit signed displacement.
   a. 8
   b. 16
   c. 32
   d. None of the above
2. The range for the short jump instructions from the IP of the next instruction is
   a. 0 to 255
   b. −128 to +127
   c. −32768 to +32767
   d. None of the above
3. Which of the following needs to change only IP?
   a. Intrasegment jump
   b. Intersegment jump
   c. Both (a) and (b)
   d. None of the above
4. Which of the following instructions uses 5 bytes including opcode, CS and IP?
   a. Intrasegment direct
   b. Intrasegment indirect
   c. Intersegment direct
   d. Intersegment indirect
5. Which of the following is not a short jump instruction?
   a. JCXZ
   b. LOOPE
   c. JC
   d. All of the above are short jump instructions
6. Which of the following instructions skip the block of instructions when CX = 0?
   a. JCXZ
   b. LOOP
   c. LOOPE/LOOPZ
   d. LOOPNE/LOOPNZ
7. Which of the following instructions work before the arithmetic operation?
   a. AAA
   b. AAS
   c. AAM
   d. AAD
8. The range for an 8-bit packed BCD number in decimal is
   a. 0 to 9
   b. 0 to 99
   c. 0 to 100
   d. None of the above
9. Which of the following flags cannot set or reset by an instruction?
   a. Carry flag
   b. Direction flag
   c. Trap flag
   d. None of the above
10. Which of the following hardware synchronization instructions is used as prefix to another instruction?
    a. WAIT
    b. LOCK
    c. HLT
    d. TEST

11. Which of the following hardware synchronization instructions is used to communicate with the coprocessor?
    a. WAIT
    b. LOCK
    c. HLT
    d. TEST

12. Which of the following instructions is used as time delay?
    a. HLT
    b. LOCK
    c. NOP
    d. None of the above

Review Questions

1. What is unconditional jump? Differentiate between the forward jump and the backward jump with example.
2. Differentiate between the short jump and the near jump.
3. Explain with example how the IP of the target instruction is computed in the case of intrasegment direct jump instruction.
4. Differentiate between the direct and indirect intrasegment jump instructions.
5. Differentiate between the direct and indirect intersegment jump instructions.
6. Describe the intrasegment and intersegment jump instructions.
7. Differentiate between the near and far jump instructions.
8. Discuss the conditional jump instructions used to deal with unsigned and signed numbers.
9. What problem does the assembler face while coding the forward jump? How does it solve the problem?
10. What is the meaning of assembler directive SHORT? Give its use.
11. What is decision making? How can you implement it in the 8086 programming? Explain with proper example.
12. Explain the implementation of looping in the 8086 programming with example.
13. What are the loop instructions? Explain their working with example.
14. Explain the working of the LOOPE/LOOPZ instruction with example.
15. What is the JCXZ instruction? What is its use?
16. Explain the working of the AAA instruction with example.
17. What is the DAA instruction? Show its use with example.
18. Explain the working of the following instructions with example: AAM, DAS, AAD.
19. What are the processor control instructions? List them with their brief meaning.
20. How can you set the trap flag? Provide the necessary instruction sequence.
21. What is NOP instruction? Show its use for creating the delay loops.
22. List the external hardware synchronization instructions and explain their meaning.
23. Give the functions of LOCK and TEST instructions.
24. Write short notes on the following:
    (a) Intrasegment vs intersegment jumps
    (b) Conditional jump instructions
    (c) Loop instructions
    (d) Processor control instructions
    (e) BCD arithmetic in 8086

Programming Problems

1. Consider the following data segment definition:

```
data  SEGMENT
      n1 DD 12D2785Ah
      n2 DD 56DA67F1h
      ans DD ?
data  ENDS
```

   Now write a program to find the minimum from n1 and n2 and store the result into variable ans.

   (**Hint**: First compare the upper word of both the numbers and if they are equal then only the lower words need to be compared.)

2. Write a program to count the number of zero values from a block of *N* 16-bit numbers.
3. Write a program to add only odd numbers from a block of ten 16-bit numbers.
4. Write a program to add a block of *N* 16-bit numbers and get the 32-bit result into the DX:AX register pair.
5. Write a program which exchanges the minimum and maximum numbers of a given array of ten 16-bit numbers.
6. Write a program to sort the given array of 8-bit numbers using bubble sort. The bubble sort moves light values up in each pass.
7. Write a program to search a 16-bit number in an array of 16-bit numbers. Store the position of the number in the AL register, otherwise store −1.
8. Write a program to count from 0 to 50 in decimal using the DAA instruction. Store the count sequence in the data segment as an array of 50 decimal numbers as count.
9. Write a program to multiply decimal numbers 7 and 8 and store its answer as packed BCD number 56 in the data segment.

Answers

True or False. Reasons are left as an exercise

1. False
2. True
3. True
4. False
5. True
6. True
7. False
8. True
9. True
10. False
11. True
12. True

Multiple-Choice Questions

1. (b)
2. (b)
3. (a)
4. (c)
5. (d)
6. (a)
7. (d)
8. (b)
9. (c)
10. (b)
11. (a)
12. (c)