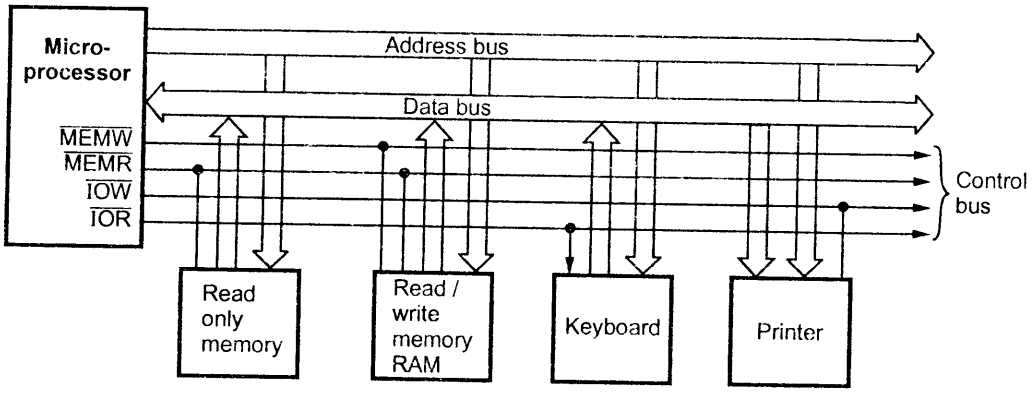


Fig. 1.28 Bus structure of microprocessor based computer system

Microprocessor	Data bus width	Address bus width	Memory size
8086	16	20	1M
8088	8	20	1M
80186	16	20	1M
80188	8	20	1M
80286	16	24	16M
80386SX	16	24	16M
80386DX	32	32	4GB
80486	32	32	4G
Pentium	64	32	4G
Pentium Pro	64	36	64K
Pentium overdrive	32	32	4G
Pentium II	64	32	64G
Pentium II, III, IV	64	36	64G

Table 1.8

The data bus lines are bi-directional. This means that microprocessor can read data in on these lines from memory or from a port, as well as send data out on these lines to a memory location or to a port. The data bus is connected in parallel to all peripherals. The

communication between peripheral and microprocessor is activated by giving output enable pulse to the peripheral. Outputs of peripherals are floated or tri-stated when they are not in use.

2) Address bus : It is an unidirectional bus. The address bus consists of 16, 20, 24 or more parallel signal lines. On these lines the microprocessor sends the address of the memory location or I/O port that is to be written to or read from.

3) Control bus : The control lines regulate the activity on the bus. The microprocessor sends signals on the control bus to enable the outputs of addressed memory devices or port devices. Typical control bus signals are memory read ($\overline{\text{MEMR}}$), memory write ($\overline{\text{MEMW}}$), I/O read ($\overline{\text{IOR}}$) and I/O write ($\overline{\text{IOW}}$).

The signals on all these buses must be co-ordinated with the signals created by various components connected to the bus. The co-ordination between these signals is monitored by bus **control unit**.

1.3.4 Operating System

An operating system performs resource management and provides an interface between the user and the machine. A resource may be the microprocessor, memory, or an I/O device. Basically, an operating system is a collection of system programs that tells the machine what to do under a variety of conditions. Major operating system functions include efficient sharing of memory, I/O peripherals, and the microprocessor among several users. Along with DOS, UNIX and WINDOWS are the popular operating systems used today.

Review Questions

1. *Discuss the mechanical age as a historical background of a computer system.*
2. *Draw and explain the structure of Babbage's analytical engine.*
3. *Discuss the electrical age as a historical background of a computer system.*
4. *Discuss the programming advancements in a computer system.*
5. *Discuss the microprocessor age as a historical background of a computer system.*
6. *Give the comparison between pentium processors.*
7. *Draw and explain the block diagram of simple microprocessor based system.*
8. *Write a short note on terminologies used in microprocessor.*
9. *What do you mean by word length ?*
10. *Explain different phases in the execution processes.*
11. *Explain fetching, decoding and execution operations of microprocessor.*
12. *How many address lines are required to access 2 MB of memory ?*
13. *What is stack ? What do you mean by stack pointer ?*
14. *What is the function of 'Timing and control unit' in microprocessor ?*
15. *Which are the different types of buses used in microprocessor ?*

16. *Write a short note on microprocessor based personal computer system.*
17. *Draw the map of memory system of microprocessor based personal computer system.*
18. *Give the logical grouping of system memory. What is conventional memory ? Where it is located ? What is its purpose ?*
19. *Draw and explain the first 1 Mbyte memory work space.*
20. *What is extended memory ? Draw and explain the memory map for conventional and extended memory.*
21. *What is expanded memory ? Draw and explain the memory map for conventional, extended and expanded memories.*
22. *How expanded memory is accessed ?*
23. *Write short notes on :*
    a) Extended memory
    b) Conventional memory
    c) Upper memory area
    d) High memory area
    e) Expanded memory
24. *What is an operating system?*

❑❑❑

2

8086 / 8088 CPU

In 1978, Intel came out with the 8086 processor. The Intel 8086 is a 16-bit microprocessor, implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged it in a 40 pin dual in line package. In this chapter, we study features, architecture, register organization, bus operation and memory segmentation.

2.1 Features of 8086

1. The 8086 is a 16-bit microprocessor. The term "16-bit" means that its arithmetic logic unit, internal registers and most of its instructions are designed to work with 16-bit binary words.
2. The 8086 has a 16-bit data bus, so it can read data from or write data to memory and ports either 16 bits or 8 bits at a time. The 8088, however, has an 8-bit data bus, so it can only read data from or write data to memory and ports 8 bits at a time.
3. The 8086 has a 20-bit address bus, so it can directly access 2^{20} or 10,48,576 (1 Mb) memory locations. Each of the 10, 48, 576 memory locations is byte wide. Therefore, a sixteen-bit words are stored in two consecutive memory locations. The 8088 also has a 20-bit address bus, so it can also address 2^{20} or 10, 48, 576 memory locations.
4. The 8086 can generate 16-bit I/O address, hence it can access 2^{16} = 65536 I/O ports.
5. The 8086 provides fourteen 16-bit registers.
6. The 8086 has multiplexed address and data bus which reduces the number of pins needed, but does slow down the transfer of data (drawback).
7. The 8086 requires one phase clock with a 33 % duty cycle to provide optimized internal timing.

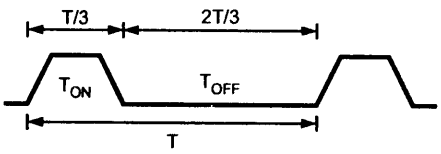


Fig. 2.1 Clock cycle

Range of clock rates are :- 5 MHz for 8086
8 MHz for 8086-2
10 MHz for 8086-1

8. The 8086 is possible to perform bit, byte, word and block operations in 8086. It performs the arithmetic and logical operations on bit, byte, word and decimal numbers including multiply and divide.
9. The Intel 8086 is designed to operate in two modes, namely the minimum mode and the maximum mode. When only one 8086 CPU is to be used in a microcomputer system, the 8086 is used in the minimum mode of operation. In this mode the CPU issues the control signals required by memory and I/O devices. In multiprocessor (more than one processor in the system) system 8086 operates in maximum mode. In maximum mode, control signals are generated with the help of external bus controller (8288).
10. The Intel 8086 supports multiprogramming. In multiprogramming, the code for two or more processes is in memory at the same time and is executed in a time-multiplexed fashion.
11. An interesting feature of the 8086 is that it fetches upto six instruction bytes (4 instruction bytes for 8088) from memory and queue stores them in order to speed up instruction execution. Later we will discuss this in detail.
12. The 8086 provides powerful instruction set with the following addressing modes : Register, immediate, direct, indirect through an index or base, indirect through the sum of a base and an index register, relative and implied.

2.2 Architecture of 8086

Fig. 2.2 shows a block diagram of the 8086 internal architecture. It is internally divided into two separate functional units. These are the Bus Interface Unit (BIU) and the Execution Unit (EU). These two functional units can work simultaneously to increase system speed and hence the throughput. Throughput is a measure of number of instructions executed per unit time.

2.2.1 Bus Interface Unit [BIU]

The bus interface unit is the 8086's interface to the outside world. It provides a full 16-bit bi-directional data bus and 20-bit address bus. The bus interface unit is responsible for performing all external bus operations, as listed below.

Functions of Bus Interface Unit

1. It sends address of the memory or I/O.
2. It fetches instruction from memory.
3. It reads data from port/memory.
4. It writes data into port/memory.
5. It supports instruction queuing.
6. It provides the address relocation facility.

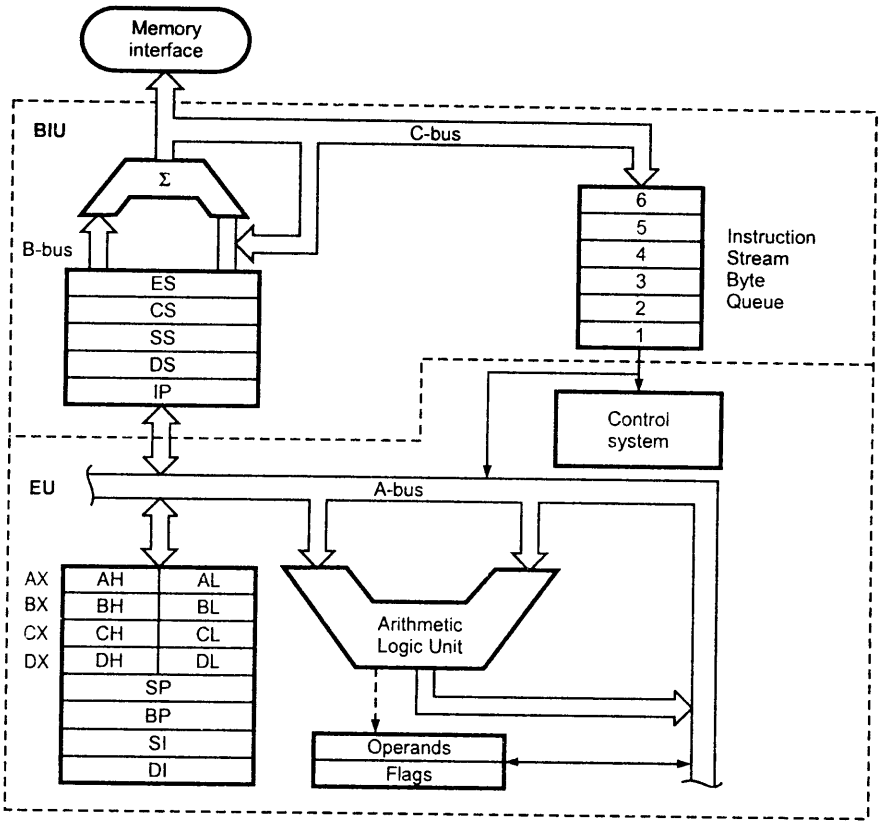


Fig. 2.2 8086 internal block diagram

To implement these functions the BIU contains the instruction queue, segment registers instruction pointer, address summer and bus control logic.

Instruction Queue

To speed up program execution, the BIU fetches **six instruction bytes** ahead of time from the memory. These prefetched instruction bytes are held for the execution unit in a group of registers called **Queue**. With the help of queue it is possible to fetch next instruction when current instruction is in execution. For example, current instruction in execution is a multiplication instruction. In 8086, operands for multiplication operations are within registers. Still it requires 100 clock cycles to execute multiply instruction. Like multiplication there are number of other instructions in 8086 which need quite a large number of clock cycles for execution. During this execution time the BIU fetches the next instruction or instructions from memory into the instruction queue instead of remaining idle. The BIU continues this process as long as the queue is not full. Due to this, execution unit gets the ready instruction in the queue and instruction fetch time is eliminated. This is illustrated in Fig. 2.3.

The queue operates on the principle first in first out (FIFO). So that the execution unit gets the instructions for execution in the order they are fetched. In case of JUMP and CALL instructions, instruction already fetched in queue are of no use. Hence, in these

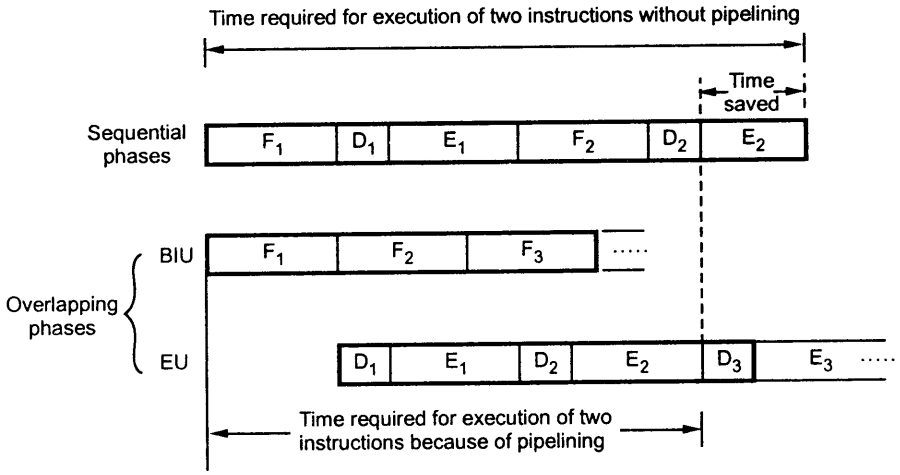


Fig. 2.3 Pipelining

cases queue is dumped and newly formed by loading instructions from new address specified by JUMP or CALL instruction. Feature of fetching the next instruction while the current instruction is executing is called **pipelining.**

The length of the queue should be such that EU should get the next instruction from the queue of the BIU immediately after the execution of the current instruction. To satisfy this, number of pre-fetched instruction in the queue and hence the queue length depends on the fetching speed and the execution speed. Sometime queue length may be restricted due to the space available on the CPU chip.

2.2.2 Execution Unit [EU]

The execution unit of 8086 tells the BIU from where to fetch instructions or data, decodes instructions and executes instructions. It contains

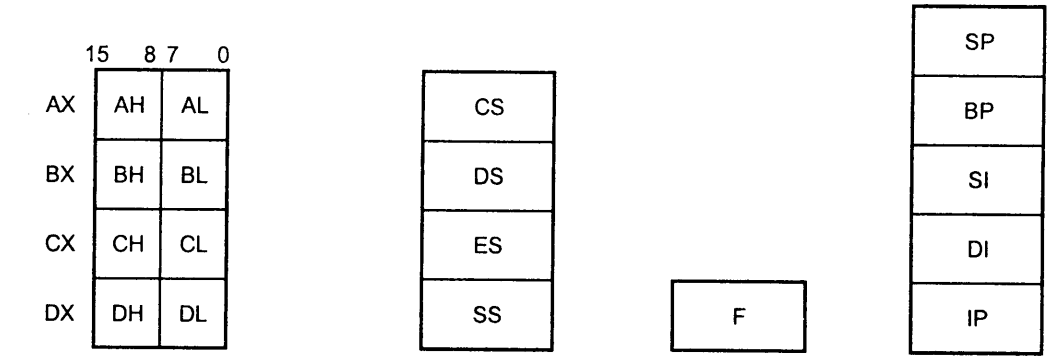
- Control circuitry
- Instruction decoder
- Arithmetic logic unit (ALU)
- Register organisation
  - Flag register
  - General purpose registers
  - Pointers and index registers

Control circuitry, Instruction decoder, ALU

The control circuitry in the EU directs the internal operations. A decoder in the EU translates the instructions fetched from memory into a series of actions which the EU performs. ALU is 16-bit. It can add, subtract, AND, OR, XOR, increment, decrements, complement and shift binary numbers.

2.3 Register Organization

The 8086 has a powerful set of registers. It includes general purpose registers, segment registers, pointers and index registers, and flag register. The Fig. 2.4 shows the register organization of 8086. It is also known as programmer's model of 8086. The registers shown in programmer's model are accessible to programmer. As shown in the Fig. 2.4, all the registers of 8086 are 16-bit registers.



(a) General purpose registers (b) Segment registers (c) Flag registers (d) Pointer and index registers

Fig. 2.4 Register organization of 8086

2.3.1 General Purpose Registers

The 8086 has four 16-bit general purpose registers labeled AX, BX, CX and DX. Each 16-bit general purpose register can be split into two 8-bit registers. The letters L and H specify the lower and higher bytes of a particular register. For example, BH means the higher byte (8-bits) of the BX register and BL means the lower byte (8-bits) of the BX register. The letter X is used to specify the complete 16-bit register.

The general purpose registers are either used for holding data, variables and intermediate results temporarily. They can also be used as a counters or used for storing offset address for some particular addressing modes. The register AX is used as 16-bit accumulator whereas register AL (lowerbyte of AX) is used as 8-bit accumulator. The register BX is also used as offset storage for generating physical addresses in case of certain addressing modes. On the other hand, the register CX is also used as a default counter in case of string and loop instructions.

2.3.2 Segment Registers

The physical address of the 8086 is 20-bits wide to access 1 Mbyte memory locations. However, its registers and memory locations which contain logical addresses are just 16-bits wide. Hence 8086 uses memory segmentation. It treats the 1 Mbyte of memory as divided into segments, with a maximum size of a segment as 64 kbytes. Thus any location within the segment can be accessed using 16-bits. The 8086 allows only four active

segments at a time, as shown in the Fig. 2.5. For the selection of the four active segments the 16-bit segment registers are provided by the bus interface unit (BIU) of the 8086. These four registers are :

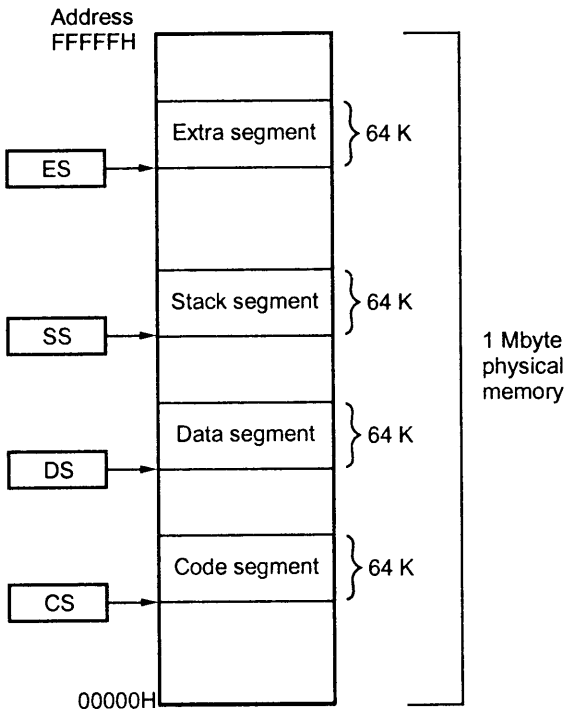


Fig. 2.5 Memory segmentation and segment registers

Code segment (CS) register, the data segment (DS) register, the stack segment (SS) register, and the extra segment (ES) register. These are used to hold the upper 16-bits of the starting addresses of the four memory segments, on which 8086 works at a particular time. For example, the value in CS identifies the starting address of 64 kbyte segment known as code segment. By "starting address", we mean the lowest addressed byte in the active code segment. The starting address is also known as **base address** or **segment base**.

The BIU always inserts zeros for the lower 4 bits (nibble) in the contents of segment register to generate 20-bit base address. For example, if the code segment register contains 348AH, then code segment will start at address 348A0H.

Functions of Segment Registers

1. The CS register holds the upper 16-bits of the starting address of the segment from which the BIU is currently fetching the instruction code byte.
2. The SS register is used for the upper 16-bits of the starting address for the program stack (all stack related instructions will operate on stack).

3. ES register and DS register are used to hold the upper 16-bits of the starting address of the two memory segments which are used for data.

2.3.3 Pointers and Index Registers

All segment registers are 16-bit wide. But it is necessary to generate 20-bit address (physical address) on the address bus. To get 20-bit physical address one or more pointer or index registers are associated with each segment register. The pointer registers IP, BP and SP are associated with code, data and stack segments, respectively. They hold the offset within the code, data and stack segments, respectively. The index registers DI and SI are used as a general purpose registers as well as for offset storage in case of indexed, based indexed and relative based indexed addressing modes. The detail description of pointers and index register is given in section 2.5.

2.3.4 Flag Register

A flag is a flip-flop which indicates some condition produced by the execution of an instruction or controls certain operations of the EU. The flag register contains nine active flags as shown in the Fig. 2.6.

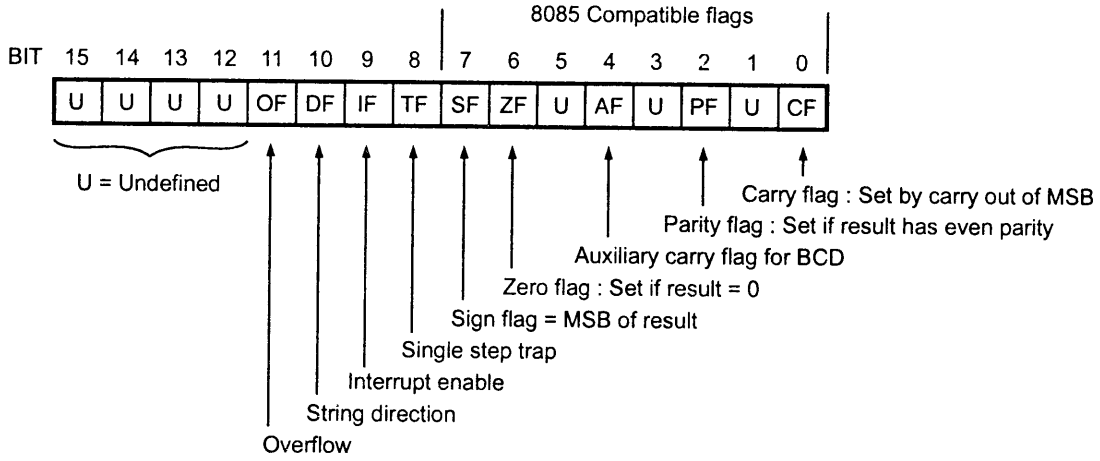


Fig. 2.6 8086 flag register bit pattern

Six of them are used to indicate some condition produced by instruction.

1. **Carry Flag (CF) :** In case of addition this flag is set if there is a carry out of the MSB. The carry flag also serves as a borrow flag for subtraction. In case of subtraction it is set when borrow is needed.

2. **Parity Flag (PF) :** It is set to 1 if result of byte operation or lower byte of the word operation contain an even number of ones; otherwise it is zero.

3. **Auxiliary Flag (AF):** This flag is set if there is an overflow out of bit 3 i.e., carry from lower nibble to higher nibble (D_3 bit to D_4 bit). This flag is used for BCD operations and it is not available for the programmer.

4. Zero Flag (ZF) : The zero flag sets if the result of operation in ALU is zero and flag resets if the result is nonzero. The zero flag is also set if a certain register content becomes zero following an increment or decrement operation of that register.

5. Sign Flag (SF): After the execution of arithmetic or logical operations, if the MSB of the result is 1, the sign bit is set. Sign bit 1 indicates the result is negative; otherwise it is positive.

6. Overflow Flag (OF): This flag is set if result is out of range. For addition this flag is set when there is a carry into the MSB and no carry out of the MSB or vice-versa. For subtraction, it is set when the MSB needs a borrow and there is no borrow from the MSB, or vice-versa.

Example 2.1 : *Give the contents of the flag register after execution of following addition.*

$$\begin{array}{r} 0110\ 0101\ 1101\ 0001 \\ +\ 0010\ 0011\ 0101\ 1001 \\ \hline 1000\ 1001\ 0010\ 1010 \end{array}$$

Solution : SF = 1, ZF = 0, PF = 1, CF = 0, AF = 0 , OF = 1

Example 2.2 : *Give the contents of the flag register after execution of following subtraction.*

$$\begin{array}{r} 0110\ 0111\ 0010\ 1001 \\ -\ 0011\ 0101\ 0100\ 1010 \\ \hline 0011\ 0001\ 1101\ 1111 \end{array}$$

Solution : SF = 0, ZF = 0, PF = 1, CF = 0, AF = 1, OF = 0

The three remaining flags are used to control certain operations of the processor.

1. Trap Flag (TF): One way to debug a program is to run the program one instruction at a time and see the contents of used registers and memory variables after execution of every instruction. This process is called **'single stepping'** through a program. Trap flag is used for single stepping through a program. If set, a trap is executed after execution of each instruction, i.e. interrupt service routine is executed which displays various registers and memory variable contents on the display after execution of each instruction. Thus programmer can easily trace and correct errors in the program.

2. Interrupt Flag (IF) : It is used to allow/prohibit the interruption of a program. If set, a certain type of interrupt (a maskable interrupt) can be recognized by the 8086; otherwise, these interrupts are ignored.

3. Direction Flag (DF) : It is used with string instructions. If DF = 0, the string is processed from its beginning with the first element having the lowest address. Otherwise, the string is processed from the high address towards the low address.

2.4 Bus Operation

The 8086 has a common address and data bus. The address and data are time multiplexed, i.e. address and data appear on this bus at different time intervals. Thus bus is commonly known as multiplexed address and data bus. The multiplexed address and data bus provides the most efficient use of pins on the processor while permitting the use of a standard 40-lead package. This multiplexed address and data bus has to be demultiplexed externally with the use of latches and the ALE signal provided by 8086. This bus can be buffered directly and used throughout the system with address latching provided on memory and I/O modules or it can be demultiplexed at the processor with a single set of address latches if a standard non-multiplexed bus is desired for the system.

The control operation of 8086 is different in two different modes : minimum mode and maximum mode. The 8086 provides some signals which have different meanings in minium mode and maximum mode. The minimum mode is used for a small systems with a single processor and maximum mode is for medium size to large systems, which often include two or more processors.

2.5 Memory Segmentation

Two types of memory organisations are commonly used. These are **linear addressing** and **segmented addressing**. In linear addressing the entire memory space is available to the processor in one linear array. In the segmented addressing, on the other hand, the available memory space is divided into "chunks" called segments. Such a memory is known as **segmented memory**. In 8086 system the available memory space is 1 Mbytes. This memory is divided into number of logical segments. Each segment is 64 kbytes in size and addressed by one of the segment registers. The 16-bit contents of the segment register gives the starting/base address of a particular segment, as shown in Fig. 2.7. To address a specific memory location within a segment we need an offset address. The offset address is also 16-bit wide and it is provided by one of the associated pointer or index register.

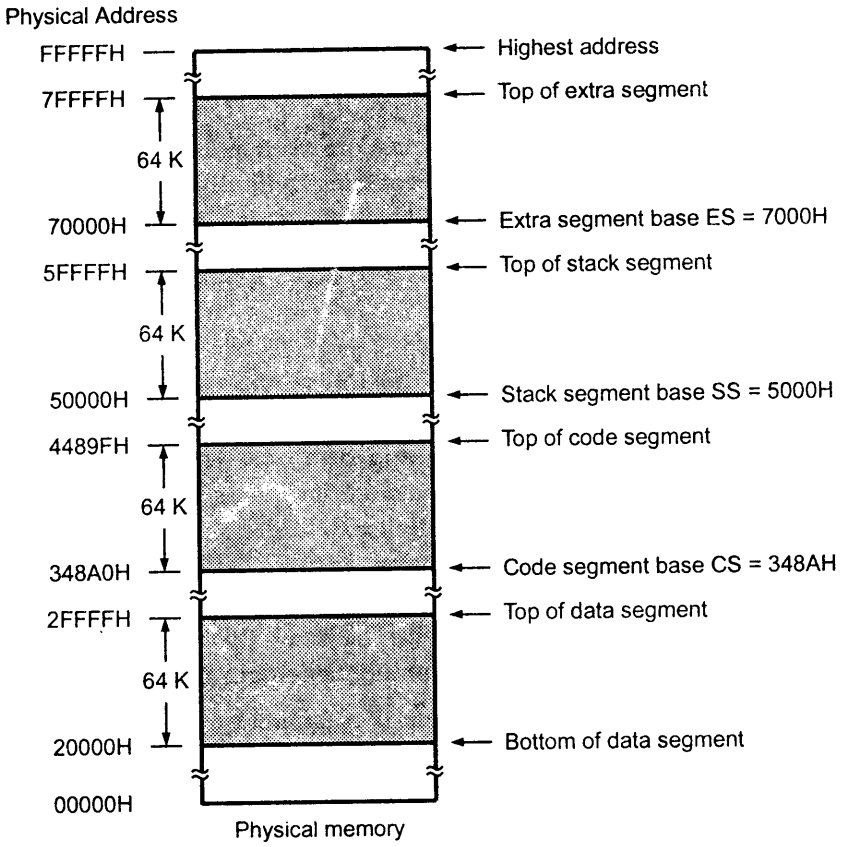


Fig. 2.7 Memory segmentation

Rules for Memory Segmentation

1. The four segments can overlap for small programs. In a minimum system all four segments can start at the address 00000H.
2. The segment can begin/start at any memory address which is divisible by 16.

Advantages of Memory Segmentation

1. It allows the memory addressing capacity to be 1 Mbyte even though the address associated with individual instruction is only 16-bit.
2. It allows instruction code, data, stack, and portion of program to be more than 64 kB long by using more than one code, data, stack segment, and extra segment.
3. It facilitates use of separate memory areas for program, data and stack.
4. It permits a program or its data to be put in different areas of memory, each time the program is executed i.e. program can be relocated which is very useful in multiprogramming.

Generation of 20-bit Address

To access a specific memory location from any segment we need 20-bit physical address. The 8086 generates this address using the contents of segment register and the offset register associated with it. Let us see how 8086 access code byte within the code segment.

We know that the CS register holds the base address of the code segment. The 8086 provides an instruction pointer (IP) which holds the 16-bit address of the next code byte within the code segment. The value contained in the IP is referred to as an **offset**. This value must be offset from (added to) the segment base address in CS to produce the required 20-bit physical address.

The contents of the CS register are multiplied by 16. i.e. shifted by 4 position to the left by inserting 4 zero bits and then the offset i.e. the contents of IP register are added to the shifted contents of CS to generate physical address. As shown in the Fig. 2.8, the contents of CS register are 348AH, therefore the shifted contents of CS register are 348A0H. When the BIU adds the offset of 4214H in the IP to this starting address, we get 38AB4H as a 20-bit physical address of memory. This is illustrated in Fig. 2.8 (b).

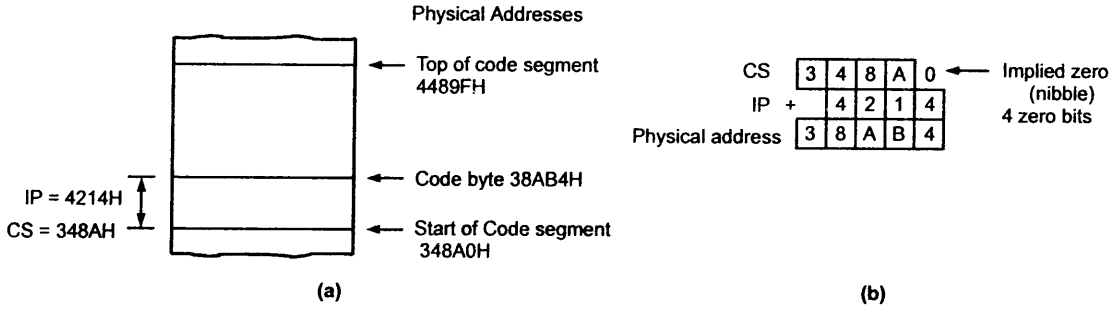


Fig. 2.8

We have seen that how 20-bit physical address is generated within the code segment. In the similar way the 20-bit physical address is generated in the other segments. However, it is important to note that each segment requires particular segment register and offset register to generate 20-bit physical address.

Pointers and Index Registers

All segment registers are 16-bit. But it is necessary to put 20-bit address (physical address) on the address bus. To get 20-bit physical address one more register is associated with each segment register the way IP is associated with CS.

These additional registers belong to the pointer and index group. The pointer and index group consists of instruction pointer (IP), stack pointer (SP), BP (base pointer), source index (SI) and destination index (DI) registers.

Stack Pointer (SP) : The stack pointer (SP) register contains the 16-bit offset from the start of the segment to the top of stack. For stack operation, physical address is produced by adding the contents of stack pointer register to the segment base address in SS. To do this the contents of the stack segment register are shifted four bits left and the contents of SP are added to the shifted result. If the contents of SP are 9F20H and SS are 4000H then the physical address is calculated as follows.

SS = 4000H after shifting four bits left SS = 40000H

Now

SS	40000H
+ SP	9F20H
Physical address	49F20H

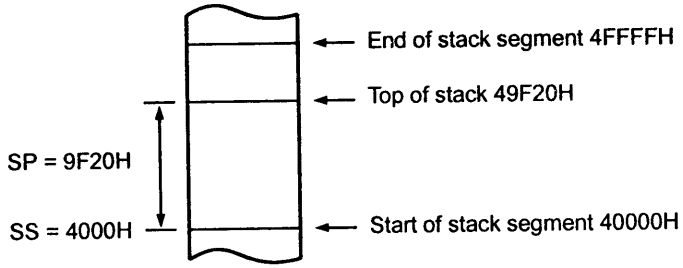


Fig. 2.9 Stack and stack pointer

Base Pointer, Source Index and Destination Index (BP, SI and DI)

These three 16-bit registers can be used as general purpose registers. However, their main use is to hold the 16-bit offset of the data word in one of the segments.

Base Pointer (BP) : We can use the BP register instead of SP for accessing the stack using the based addressing mode. In this case, the 20-bit physical stack address is calculated from BP and SS. Addressing modes are discussed in later section.

Source Index (SI) : Source index (SI) can be used to hold the offset of a data word in the data segment. In this case, the 20-bit physical data address is calculated from SI and DS.

Destination Index (DI) : The ES register points to the extra segment in which data is stored. String instructions always use ES and DI to determined the 20-bit physical address for the destination.

Default and Alternate Register Assignments

Table 2.1 shows that some memory references and their default and alternate segment definitions. For example, instruction codes can only be stored in the code segment with IP used as an offset. Similarly, for stack operations only SS and SP or BP registers can be used to give segment and offset addresses respectively. On the other hand, for accessing general data, string source, data pointed by BX and BP registers; it is possible to use alternate segments by using segment override prefix. See examples given after Table 2.1.

Type of Memory Reference	Default Segment	Alternate Segment	Offset (Logical Address)
Instruction fetch	CS	None	IP
Stack operation	SS	None	SP, BP
General data	DS	CS, ES, SS	Effective address
String source	DS	CS, ES, SS	SI
String destination	ES	None	DI
BX used as pointer	DS	CS, ES, SS	Effective address
BP used as pointer	SS	CS, ES, DS	Effective address

Table 2.1 Default and alternate register assignments

For the following examples we have assumed

CS = 1000H, DS = 2000H, SS = 3000H, ES = 4000H, BP = 0010 H,

BX = 0020H, SP = 0030H, SI = 0040H, DI = 0050H

Example :

1) MOV AL, [BP]

```
        3000[0]H   SS
+         0010 H   BP
Physical Address 30010 H
```

This instruction copies a byte from memory location to the AL register. The effective address for the memory location is contained in the BP register. By default, an effective address is added to the stack segment (SS) to produce the physical memory address (30010 H).

2) MOV CX, [BX]

```
        2000[0]H   DS
+         0020 H   BX
Physical Address 20020 H
```

This instruction copies a word from memory location to the CX register. The effective address is contained in the BX register. By default an effective address is added to the data segment (DS) to produce the physical memory address (20020 H).

3) MOV AL, [BP+SI]

```
          0010 H   BP
+         0040 H   SI
Effective Address 0050 H
```

This instruction copies a byte from memory location to the AL register. The effective address is the summation of the contents of the BP and SI register.

```
        3000[0]H   SS
+         0050 H   EA
Physical Address 30050 H
```

The effective address is added to the stack segment (SS) to get the physical address.

4) MOV CS : [BX], AL

```
    1000[0]H   CS
+     0020 H   BX
-------------------
Physical Address 1002 0 H
```

This instruction copies a byte from the AL register to a memory location. The effective address for the memory location is contained in the BX register. By default an effective address in BX will be added to the data segment (DS) to produce the physical memory address. In this instruction, the CS: in front of [BX] indicates that we want BIU to add the effective address to the code segment (CS) to produce the physical address. The CS: is called **segment override prefix.**

Segment Override Prefix

The segment override prefix allows the programmer to deviate from the default segment. The segment override prefix is an additional 8-bit code which is put in memory before the code for the rest of the instruction. This additional code selects the alternate segment register. The code byte for the segment override prefix as the format 001XX110. The XX represents a 2 bits which are as follows : ES = 00, CS = 01, SS = 10 and DS = 11. It is important to note that the segment override prefix may be added to almost any instruction in any memory addressing mode.

2.6 The Processor 8088

In 1978, Intel came out with the 8086 processor. Since it is a 16-bit processor and has a tremendous flexibility in the programming as compared to 8085, it was a remarkable step in the development of high speed computing machines. Before the introduction of 8086, most of the microprocessor based systems was built up with microprocessors 8080 or 8085. Therefore, the interfacing circuits for different applications was designed for these 8-bit microprocessors. So naturally, after introduction of 8086, there was a demand for a microprocessor chip which has the software compatibility with 8086 and external interface like 8085. This was the main purpose behind the design of 8088 microprocessor.

The 8088 is a 8-bit microprocessor that is fully software compatible with the 8086 (i.e. it has the same instruction set) and can be used in a hardware system that was built for an 8080 or 8085. Like the 8080 and 8085 it has 8 data lines, but its CPU architecture is essentially the same as that of the 8086 except few minor changes. The 8086 has 6 byte instruction queue whereas 8088 has 4 byte instruction queue, as shown in Fig. 2.10. The pin assignments for 8088 are the same as the 8086 except that the 8088 address pins A_{15} through A_8 are used only for addresses and one of the control pins, the high-byte enable ($\overline{BHE}$)pin, has been changed to a status pin because the 8088 can access only one byte at a time. The M/$\overline{IO}$ signal of 8086 is made IO/$\overline{M}$ in 8088 to maintain the hardware compatibility with 8085.

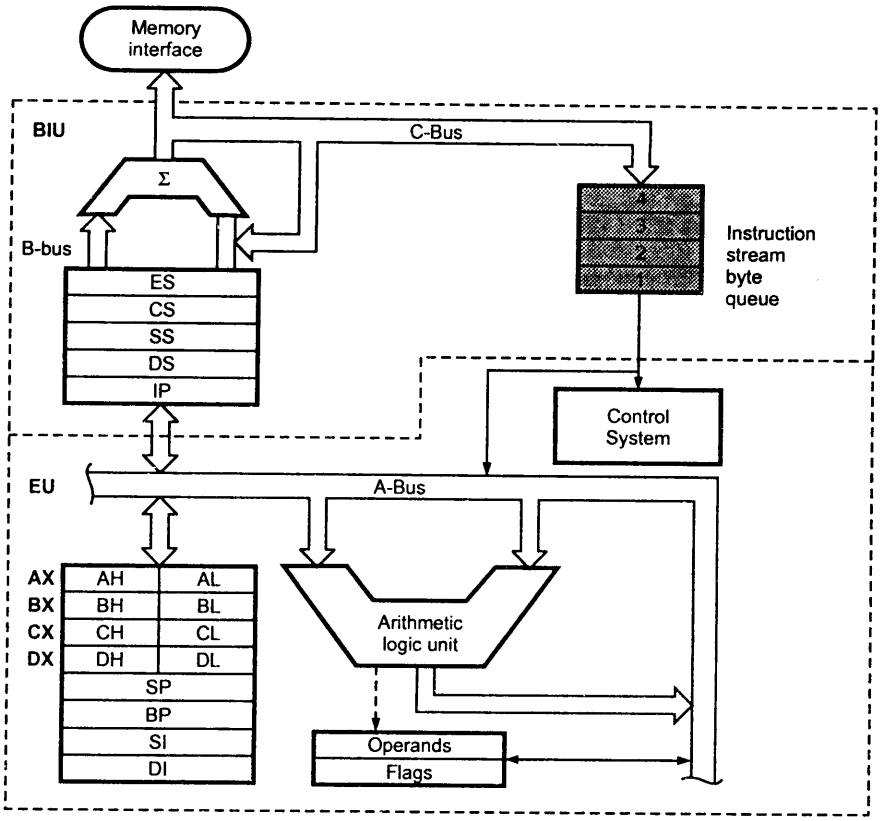


Fig. 2.10 8088 Internal block diagram

Sr. No	Microprocessor 8088	Microprocessor 8086
1.	It has only eight data lines. Therefore, it has AD_0 - AD_7 and A_8 - A_{15} signals.	It has sixteen data lines. Therefore it has AD_0 - AD_{15} signals.
2.	As data bus is 8-bit wide, it does not have $\overline{BHE}$ signal.	It has $\overline{BHE}$ signal to access higher byte.
3.	It has 4 byte instruction queue. Due to 8-bit data bus, instruction fetching is slow and 4 bytes are sufficient for queue.	It has 6 byte instruction queue.
4.	Its pin number 34 is $\overline{SS_0}$. It acts as $\overline{S_0}$ in the minimum mode. In maximum mode, $\overline{SS_0}$ pin is always high.	Its pin number 34 is $\overline{BHE}/S_7$. During T_1 (first clock cycle) $\overline{BHE}$ should be used to enable data on to the most significant byte of the data bus. During T_2, T_3 and T_4 status of this pin is logic 0. In maximum mode, 8087 monitors this pin to identify the CPU as a 8088 or a 8086, and accordingly sets its own queue length to 4 or 6 bytes.
5.	In minimum mode its pin 28 is assigned to signal IO/M	In minimum mode its pin 28 is assigned to signal $M/\overline{IO}$

Table 2.2

In short we can say that the differences between 8088 and 8086 are only in their BIU and not in EU. The execution unit (EU) is same for both. As EU is same, the programming instructions are exactly the same for each. Programs written for the 8086 can be run on the 8088 without any changes.

Review Questions

1. *List the features of 8086 microprocessor.*
2. *Explain the architecture of 8086 processor with the help of neat block diagram.*
3. *What is the function of bus interfacing unit ?*
4. *What is the instruction queue ? Explain its advantage.*
5. *What is pipelining ?*
6. *Explain the register organisation of 8086.*
7. *What are segment registers ? Explain the purpose of them.*
8. *Explain the purpose of pointers and index registers.*
9. *What is the function of flag register ?*
10. *How physical address is generated in 8086 ?*
11. *Draw the bit pattern for flag register of 8086 and explain the significance of each bit.*
12. *List the rules for memory segmentation.*
13. *What are the advantages of using memory segmentation ?*
14. *What do you mean by index registers ?*
15. *What are the functions of SI and DI registers ?*
16. *Draw the architecture of 8088 microprocessor.*
17. *What are the differences between 8086 and 8088 ?*

3 Instruction Set of 8086/8088 and Assembly Language Programming

3.1 Introduction

The 8086 instruction set includes equivalents of the 8085 instructions plus many new ones. The new instructions contain operations such as signed/unsigned multiplication and division, bit manipulation instructions, string instructions, and interrupt instructions.

The 8086 has approximately 117 different instructions with about 300 opcodes. The 8086 instruction set contains no operand, single operand, and two operand instructions. Except for string instructions which involve array operations, the 8086 instructions do not permit memory to memory operations.

In this chapter we study the addressing modes, instruction set of 8086 and assembler directives.

3.2 Addressing Modes

We have seen how the 8086 fetches code bytes from memory by generating 20-bit physical address with the help of IP and CS. We have also seen how the 8086 accesses the stack using SS and SP. In this section we will see the different ways that an 8086 can access the data. The different ways that a processor can access data are referred to as **addressing modes**.

The addressing modes of any processor can be broadly classified as :

- Data addressing modes.
- Program memory addressing modes.
- Stack memory addressing modes.

3.2.1 Data Addressing Modes

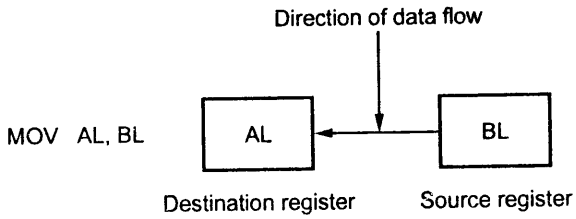
The data addressing modes can be further classified as

1. Addressing modes for accessing immediate and register data (register and immediate modes).
2. Addressing modes for accessing data in memory (memory modes).
3. Addressing modes for accessing I/O ports (I/O modes).

Addressing Modes for Accessing Immediate and Register Data

1. Register Addressing Mode

This mode specifies the source operand, destination operand, or both to be contained in an 8086 register.



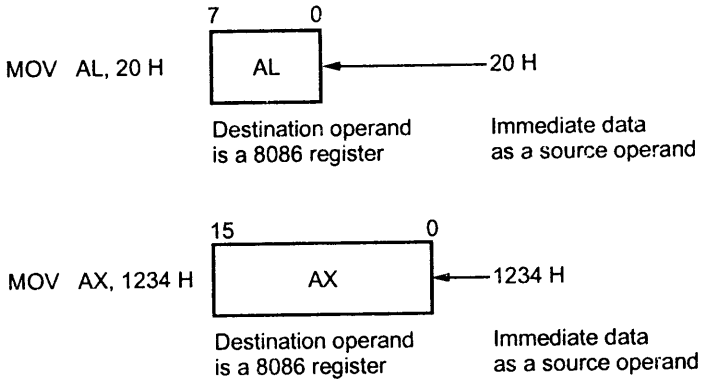
Note : Both source and destination operands are in 8086 register

Examples :

MOV BX, CX ; Copies the 16-bit contents of CX into BX
MOV CL, BL ; Copies 8-bit contents of BL into CL.

2. Immediate Addressing Mode

In an immediate mode, 8 or 16-bit data can be specified as a part of instruction.



Note : Arrow indicates direction of data flow

Examples :

MOV BL, 26H ; Copies the 8-bit data 26H into BL
MOV CX, 4567H ; Copies the 16-bit data 4567H into CX.

Addressing Modes for Accessing Data in Memory

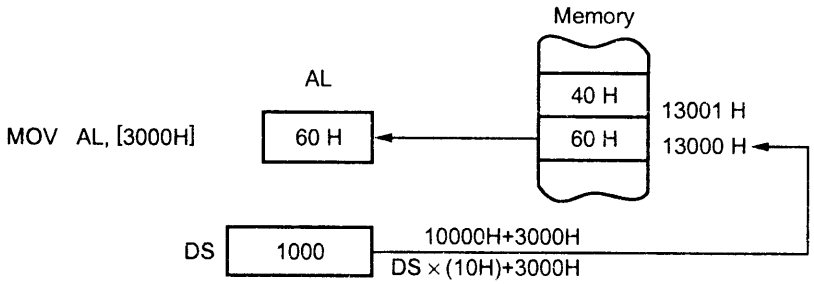
As mentioned before, the Execution Unit (EU) has direct access to all registers and data for register and immediate operands. However, the EU cannot directly access the memory operands. It must use the BIU segment registers to access memory operands. For example, when the EU needs to access a memory location, it sends an offset value to the BIU. This offset is also called the **Effective Address (EA).** Note that EA is displacement of the desired location from the segment base. As mentioned before, the BIU generates a 20-bit physical address after shifting the contents of the desired segment register four bits to the left and then adding the 16-bit EA to it.

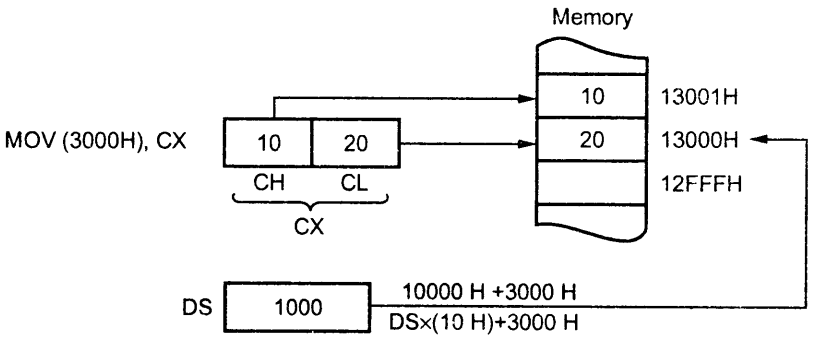
There are six ways to specify effective address (EA) in the instruction.

a. Direct addressing mode
b. Register indirect addressing mode
c. Based addressing mode
d. Indexed addressing mode
e. Based indexed addressing mode
f. String addressing mode.

1. Direct Addressing Mode :

In this mode, the 16-bit effective address (EA) is taken directly from the displacement field of the instruction. The displacement (unsigned 16-bit or sign-extended 8-bit number) is stored in the location following the instruction opcode.





Note : 1. Assume DS = 1000

∴ Physical address = DS ×(10H)+ 3000H
= 1000[0]+ 3000H = 13000H

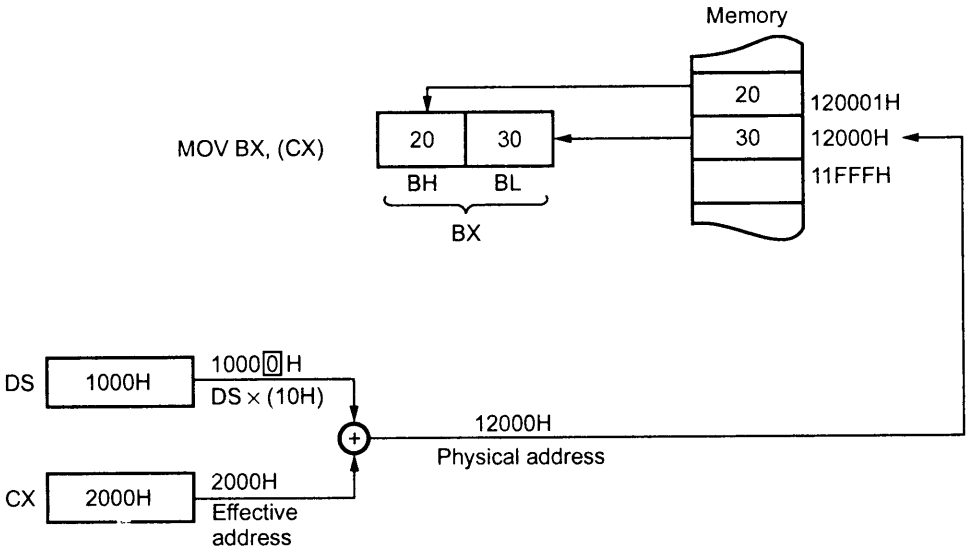
2. Arrow indicates direction of data flow.

Example :

```
MOV CL, [9823H]    ; This instruction will copy the contents of the
                   ; memory location, at a displacement of 9823H from the
                   ; data segment base, into the CL register. Here, 9823H is
                   ; the effective address (EA) which is written
                   ; directly in the instruction.
```

2. Register Indirect Addressing Mode

In this mode, the EA is specified in either a pointer register or an index register. The pointer register can be either base register BX or base pointer register BP and index register can be either Source Index (SI) register or Destination Index (DI) register. The 20-bit physical address is computed using DS and EA.



Example :

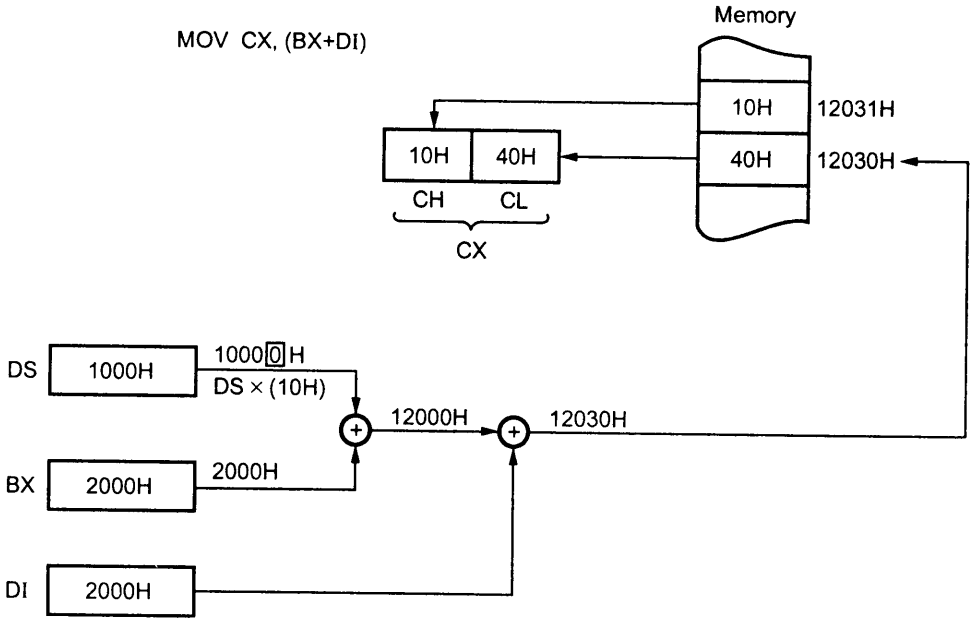
```
1. MOV [DI], BX    ; The instruction copies the 16-bit contents of BX into a
                   ; memory location offset by the value of EA specified in DI
                   ; from the current contents in DS. Now, if [DS] = 7205H,
                   ; [DI] = 0030H, and [BX] = 8765H, then after MOV [DI], BX,
                   ; content of BX (8765H) is copied to memory locations
                   ; 72080H and 72081H.
```

```
2. MOV DL, [BP]     ; This instruction copies the 8-bit
                    ; contents in DL from the memory location offset by the
                    ; value of EA specified in B P from the contents of SS.
                    ; Because data addressed by BP are by default located in
                    ; stack segment (SS).
```

3. Base-Plus-Index Addressing :

Base-plus-index addressing is similar to indirect addressing because it indirectly addresses memory data. This addressing uses one base register (BP or BX), and one index register (DI or SI) to indirectly address memory. The base register often holds the beginning location of a memory array, while the index register holds the relative position of an element in the array. Remember that whenever BP addresses the memory data, the contents of stack segment, BP and index register are used to generate physical address.

Locating Data with Base-Pulse-Index Addressing :



Locating Array Data Using Base-Plus-Index Addressing :

A main use of the base-plus-index addressing mode is to address elements in a memory array. Suppose that the array is located in the data segment beginning from memory location ARRAY. To access a particular element within the array we have to load the BX register (base) with the beginning address of the array, and the DI register (index) with the element number to be accessed. This is illustrated in Fig. 3.1.

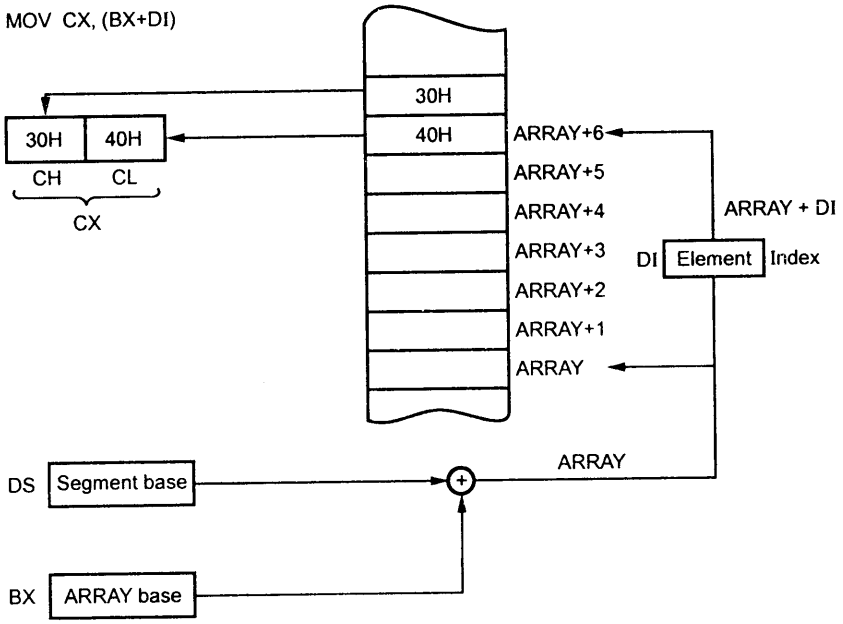


Fig. 3.1

4. Register Relative Addressing :

Register relative addressing is similar to base-plus-index addressing. Here, the data in a segment of memory are addressed by adding the displacement to the contents of a base or an index register (BP, BX, DI or SI). Remember that displacement should be added to the register within the [ ]. This is illustrated in the Fig. 3.2. Displacement can be any 8-bit or 16-bit number.

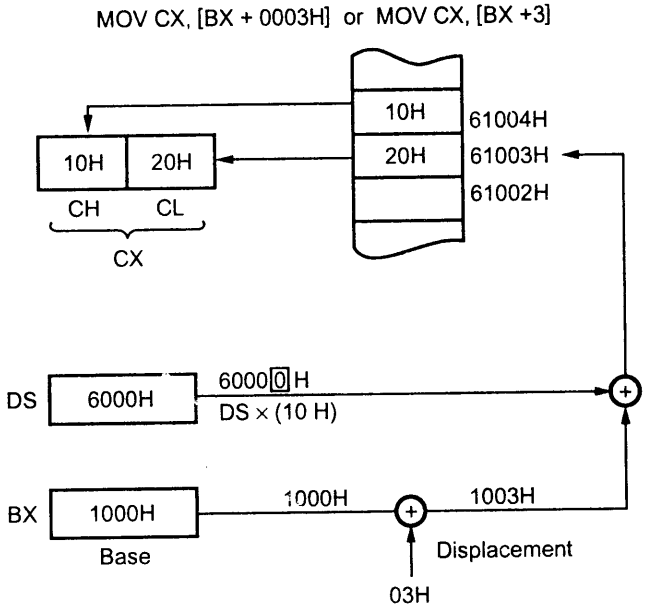


Fig. 3.2

Note :

- Displacement can be subtracted from the register : MOV AL, [DI–2].
- Displacement can be an offset address appended to the front of the [ ] : MOV AL, OFF_ADD [DI + 4].

Example : MOV AL, LAST [SI + 2] ; This instruction copies the contents of the 20-bit address computed from the displacement LAST, SI + 2 and DS into AL.

Addressing Array Data with Register Relative :

The Fig. 3.3 shows how to address data element within the array with register relative addressing.

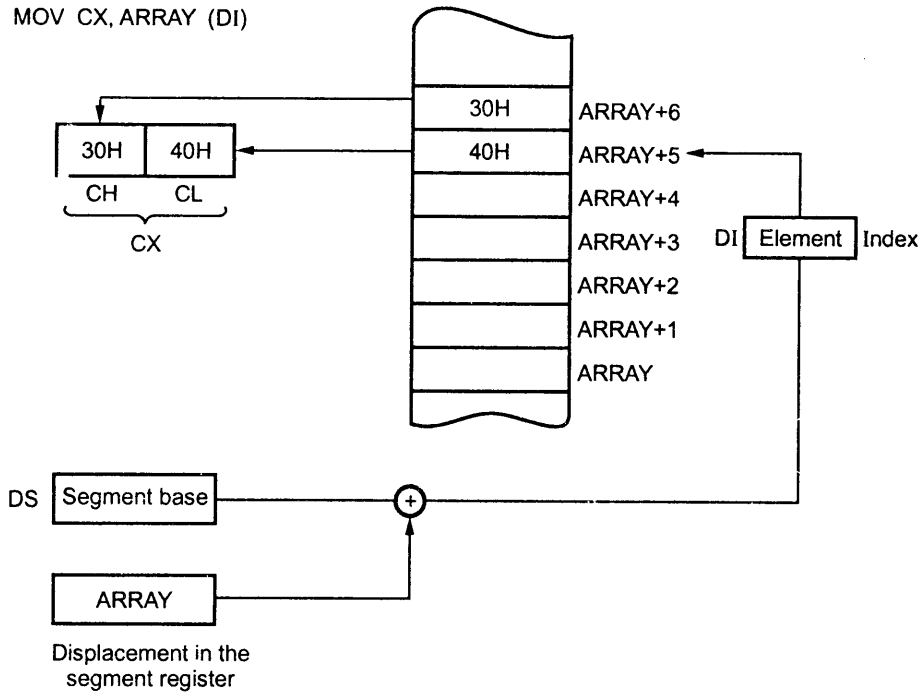


Fig. 3.3

5. Base Relative Plus Index Addressing :

The base relative plus index addressing mode is similar to the base plus index addressing mode, but it adds a displacement, besides using a base register and an index register to generate a physical address of the memory. This addressing mode is suitable to address data within the two dimensional array.

Addressing Data with Base Relative Plus Index :

The Fig. 3.4 shows how data can be accessed with base relative plus index addressing mode.

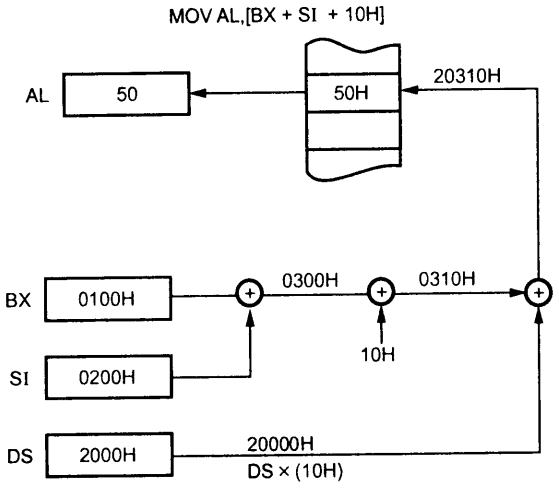


Fig. 3.4

Addressing Arrays with Base Relative-Plus-Index :

As mentioned earlier this addressing mode is useful in addressing two dimensional array. Two dimensional array usually stores records. For example, student record such as its name, roll number etc. Therefore, each record contains number of data elements. To access data element from a particular record we use base register to hold the beginning address of the array of records, index register to point a particular record in the array of records and displacement to point a particular element in the record. This is illustrated in Fig. 3.5.

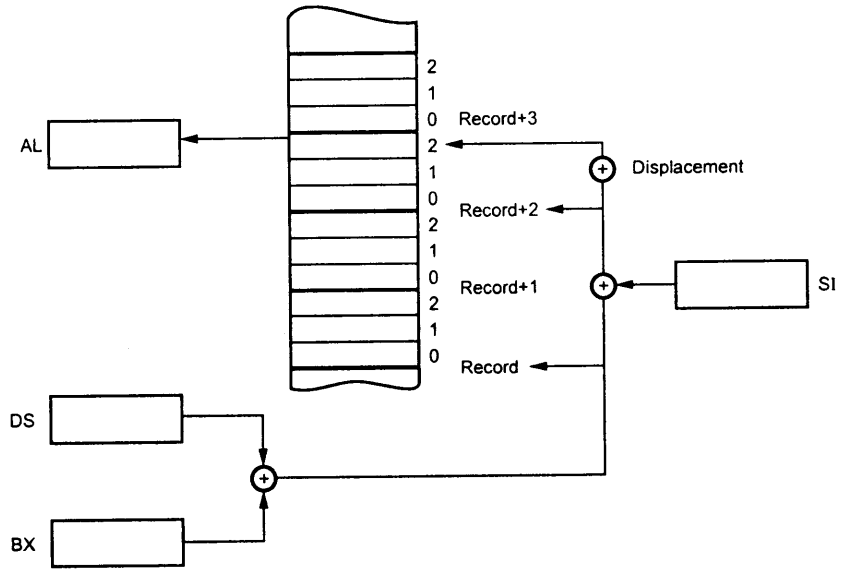


Fig. 3.5

6. String Addressing Mode :

This mode uses index registers. The string instructions automatically assume SI to point to the first byte or word of the source operand and DI to point to the first byte or word of the destination operand. The contents of SI and DI are automatically incremented (by clearing DF to 0 by CLD instruction) or decremented (by setting DF to 1 by STD instruction) to point to the next byte or word. The segment register for the source is DS. The segment register for the destination must be ES.

Example :

```
MOVS BYTE   ; If [DF] = 0, [DS] = 3000H, [SI] = 0600H, [ES] =5000H,
            ; [DI] = 0400H, [30600H] = 38H, and [50400H] = 45H, then
            ; after execution of the MOVS BYTE, [50400H] = 38H,
            ; [SI] = 0601H, and [DI] = 0401H.
```

Addressing Modes for Accessing I/O Ports (I/O Modes)

Standard I/O devices uses port addressing modes. For memory-mapped I/O, memory addressing modes are used. There are two types of port addressing modes : direct and indirect.

In direct port mode, the port number is an 8-bit immediate operand. This allows fixed access to ports numbered 0 to 255.

Example :

```
OUT 05H, AL   ; Sends the contents of AL to 8-bit port 05H.
IN AX, 80H    ; Copies 16-bit contents of port 80H
```

In indirect port mode, the port number is taken from DX allowing 64 K 8-bit ports or 32 K 16-bit ports.

Example :

```
IN AL, DX   ; If [DX] = 7890H, then it copies 8-bit content of port 7890H
            ; into AL.
IN AX, DX   ; Copies the 8-bit contents of ports 7890H and 7891H into AL
            ; and AH, respectively.
```

Note : The 8-bit and 16-bit I/O transfers must take place via AL and AX, respectively.

3.2.2 Program Memory Addressing Modes

JMP (Jump) and CALL instructions use program memory addressing modes. These instruction have three distinct forms : direct, relative and indirect. Let us see these forms and corresponding addressing modes.

Direct program memory addressing :

In this addressing mode address where to transfer program control is specified within the instruction alongwith the opcode. The Fig. 3.6 shows the direct intersegment JMP instruction and the four bytes required to store the address 20000H. This JMP instruction loads CS with 2000H and IP with 0000H to jump to memory location 20000H for the next instruction. An **intersegment jump** is a jump where destination location is from a different segment; it can be any memory location within the entire memory locations. Therefore, intersection jump is also known as **far jump.**

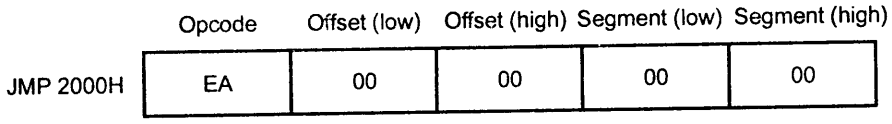


Fig. 3.6

Like JMP instruction, CALL instruction also uses direct program addressing with intersegment or far CALL instruction. Usually, in both instructions (JMP or CALL) the name of a memory address, called a **label** is specified in the instruction instead of address.

Relative program memory addressing :

In this addressing mode, the term relative is restricted to instruction pointer (IP). For example, if a JMP instruction skips the next 5 bytes of memory, the address in relation to the instruction pointer is a 5 that adds to the instruction pointer. This generates the address of the next program instruction. This is illustrated in Fig. 3.7.

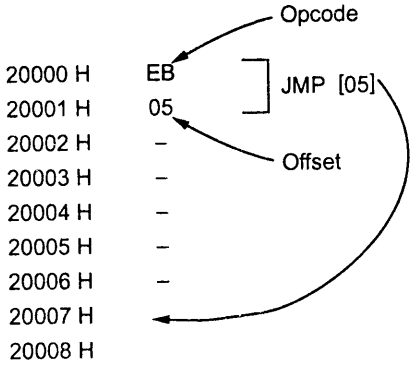


Fig. 3.7

It is important to note that in JMP instruction, opcode takes one byte and displacement may take one or two byte. When displacement is one byte (8-bit), it is called **short jump.** When displacement is two byte (16-bit), it is called **near jump.** In both (short and near) cases only contents of IP register are modified; contents of CS register are not modified. Such jumps are called **intrasegment jumps** because jumps are within the current code segment.

The relative JMP and CALL instructions can have either an 8-bit or a 16-bit signed displacement that allows a forward memory reference or a reverse memory reference.

Indirect program memory addressing :

The 8086 allows several forms of program indirect memory addressing for the JMP and CALL instructions. In this addressing mode, it is possible to use any 16-bit register (AX, BX, CX, DX, SP, BP, DI or SI); any relative register ([BP], [BX], [DI], or [SI]); and any relative register with displacement to specify the jump address. This is illustrated in Table 3.1.

Instruction	Operation
JMP BX	Jumps to memory location addressed by BX within current code segment. IP ← BX
JMP NEAR PTR [BX]	Jumps to memory location addressed by the contents of the data segment memory location addressed by BX within the current code segment. IP ← ([BX + 1], [BX]) High byte Low byte
JMP NEAR PTR [DI + 2]	Jumps to memory location addressed by the contents of the data segment memory location addressed by DI plus 2 within the current code segment. IP ← ([DI + 3], [DI + 2]) High byte Low byte
JMP ARRAY [BX]	Jumps to memory location addressed by the contents of the data segment memory location addressed by ARRAY plus BX with the current code segment. IP ← ([ARRAY + BX + 1], [ARRAY + BX]) High byte Low byte

Table 3.1

3.2.3 Stack Memory Addressing Modes

The stack is a portion of read/write memory set aside by the user for the purpose of storing information temporarily. When the information is written on the stack, the operation is called PUSH. When the information is read from stack, the operation is called a POP.

The microprocessor stores the information, much like stacking plates. Using this analogy of stacking plates it is easy to illustrate the stack operation.

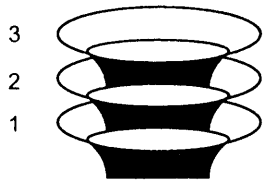


Fig. 3.8 Stacked plates

Fig. 3.8 shows the stacked plates. Here, we realize that if it is desired to take out the first stacked plate we will have to remove all plates above the first plate in the reverse order. This means that to remove first plate we will have to remove the third plate, then the second plate and finally the first plate. This means that, the first information pushed on to the stack is the last information popped off from the stack. This type of operation is known as a first in, last out (FILO). This stack is implemented with the help of special memory pointer register. The special pointer register

is called the **stack pointer.** During PUSH and POP operation, stack pointer register gives the address of memory where the information is to be stored or to be read. The stack pointer's contents are automatically manipulated to point to stack top. The memory location currently pointed by stack pointer is called **top of stack.**

Stack Structure of 8086/88

The 8086/88 has a special 16-bit register, SP to work as a stack pointer. The stack pointer (SP) register contains the 16-bit offset from the start of the segment to the top of stack. For stack operation, physical address is produced by adding the contents of stack pointer register to the segment base address in SS. To do this the contents of the stack segment register are shifted four bits left and the contents of SP are added to the shifted result. If the contents of SP are 9F20H and SS are 4000H then the physical address is calculated as follows.

SS = 4000H after shifting four bits left SS = 40000H

Now

SS	40000H
+ SP	9F20H
Physical address	49F20H

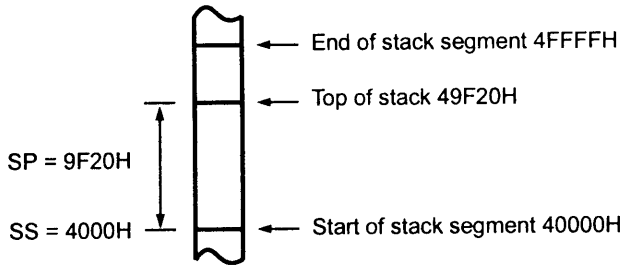


Fig. 3.9 Stack and stack pointer

PUSH and POP Operations

Temporarily stores the contents of 16-bit register or memory location or program status word, and retrieves when required. When programmer realizes the shortage of the registers, he stores the present contents of the registers in the stack with the help of PUSH instruction and then uses the registers for other function. After completion of other function programmer loads the previous contents of the register from the stack with the help of POP instruction.

PUSH Operation :

The PUSH instruction decrements stack pointer by two and copies a word from some source to the location in the stack where the stack pointer points. Here the source must be a **word** (16-bit). The source of the word can be a general purpose register, a segment register or memory. The Fig. 3.10 shows the map of the stack before and after execution of PUSH AX and PUSH CX instructions.

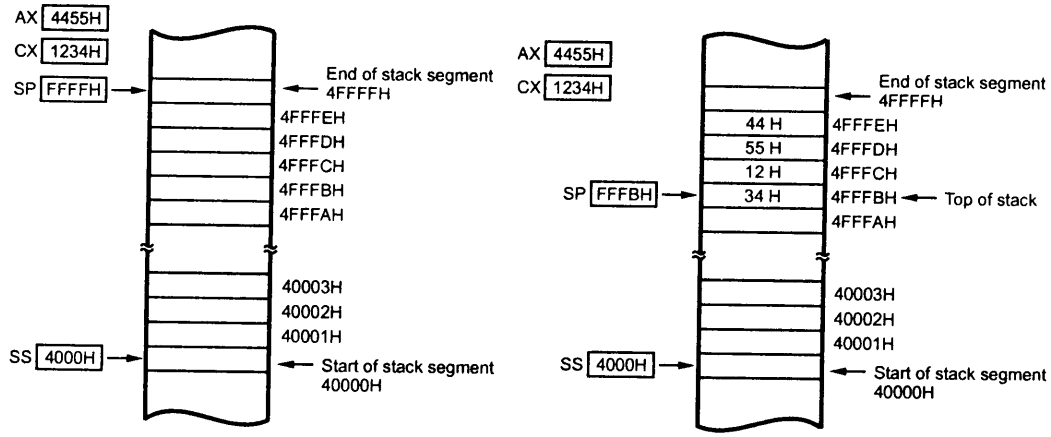


Fig. 3.10

POP Operation :

The POP instruction copies a word from the stack location pointed by the stack pointer to the destination. The destination can be a general purpose register, a segment register, or a memory location. After the word is copied to the specified destination, the stack pointer is automatically incremented by 2. The Fig. 3.11 shows the map of the stack before and after execution of POP DX and POP BX instructions.

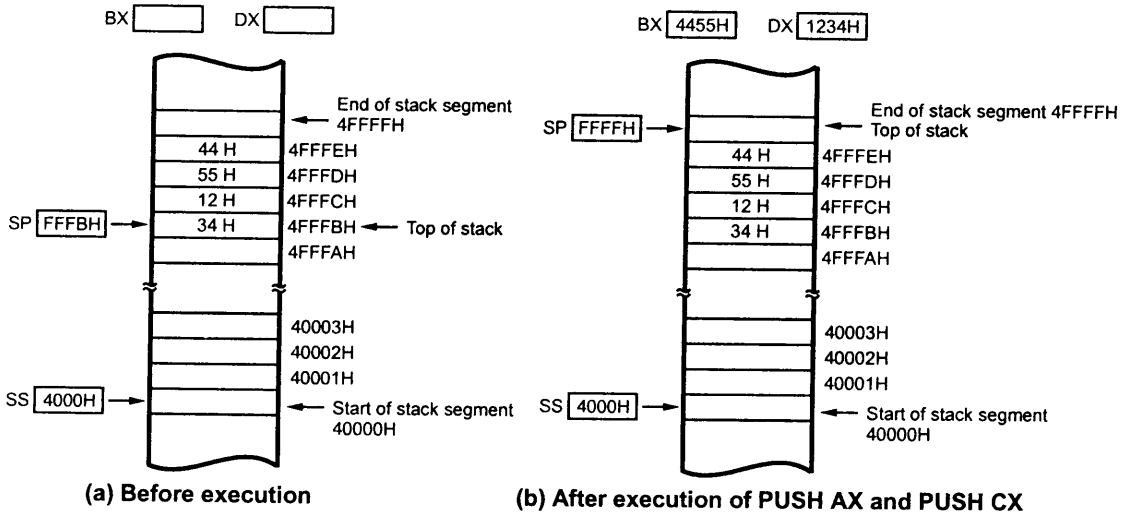


Fig. 3.11

CALL Operation

The CALL instruction is used to transfer execution to a subprogram or procedure. There are two basic types of CALLs, near and far. A near CALL is a call to a procedure which is in the same code segment as the CALL instruction. When the 8086 executes a

near CALL instruction it decrements the stack pointer by two and copies the offset of the next instruction after the CALL on the stack. It loads IP with the offset of the first instruction of the procedure in same segment.

A far CALL is a call to a procedure which is in a different segment from that which contains the CALL instruction. When the 8086 executes a far CALL it decrements the stack pointer by two and copies the contents of the CS register to the stack. It then decrements the stack pointer by two again and copies the offset of the instruction after the CALL to the stack. Finally, it loads CS with the segment base of the segment which contains the procedure and IP with the offset of the first instruction of the procedure in that segment.

RET Operation

The RET instruction will return execution from a procedure to the next instruction after the CALL instruction in the calling program. If the procedure is a near procedure (in the same code segment as the CALL instruction), then the return will be done by replacing the instruction pointer with a word from the top of the stack.

If the procedure is a far procedure (in a different code segment from the CALL instruction which calls it), then the instruction pointer will be replaced by the word at the top of the stack. The stack pointer will then be incremented by two. The code segment register is then replaced with a word from the new top of the stack. After the code segment word is popped off the stack, the stack pointer is again incremented by two. These words/word are the offset of the next instruction after the CALL. So 8086 will fetch the next instruction after the CALL.

Overflow and Underflow of Stack

We have seen the PUSH operation. During this operation stack pointer is decremented by two. We know that maximum length of stack segment is 64 K. If we go on performing PUSH operations successively, at one time the contents of SP will be 0000H. Any further attempt to PUSH data on the stack will result in **stack overflow**.

On the other hand, if we go on performing POP operations successively, at one time the contents of SP will be FFFFH. Any further attempt to POP data from the stack will result in **stack underflow**.

3.3 Instruction Set of 8086/8088

The instruction set of the 8086 is divided into Eight major groups as follows :

- Data movement instructions
- Arithmetic and logic instructions
- String instructions and
- Program control transfer instructions
- Iteration control instructions
- Processor control instructions

- External hardware synchronization instructions
- Interrupt instructions

3.4 Data Movement Instructions

The data movement instructions can be classified as :

- MOV instructions to transfer byte or word.
- PUSH/POP instructions.
- Load effective address instructions.
- String data transfer instructions.
- Miscellaneous data transfer instructions.

3.4.1 MOV Instruction

It is a general purpose instruction to transfer byte or word from register to register, register to memory or from memory to register.

MOV destination, source

The MOV instruction copies a word or a byte of data from some source to a destination. The destination can be a register or a memory location. The source can be a register, a memory location, or an immediate number. The source and destination in an instruction can't both be memory locations. The source and destination in a MOV instruction must be of same type i.e. either both must be byte or word.

MOV instruction does not affect any flags.

Examples :

```
MOV BX, 592FH          ; Load the immediate number 592FH in BX
MOV CL, [357AH]        ; Copy the contents of memory location, at a
                       ; displacement of 357AH from data segment base,
                       ; into the CL register.
MOV [734AH], BX        ; Copy the contents of BX register to two memory
                       ; locations in the data segment. Copy the contents
                       ; of BL register to memory location at a
                       ; displacement of 734AH and BH register
                       ; to memory location at a displacement of
                       ; 734BH.
MOV DS, CX             ; Copy word from CX register to data
                       ; segment register.
MOV TOTAL [BP], AX     ; Copy AX to two memory locations. AL to
                       ; first location, AH to second. Effective
                       ; address, EA, is the sum of displacement
```

```
                                ; represented by TOTAL and contents of BP.
                                ; Physical address = EA + SS.
MOV CS : TOTAL [BP], AX         ; Same as above instruction, but physical
                                ; address = EA+CS. Because the segment
                                ; overide prefix is CS.
```

3.4.2 PUSH/POP Instructions

These instructions are used to load or receive data from the stack memory.

PUSH source

The PUSH instruction decrements stack pointer by two and copies a word from some source to the location in the stack where the stack pointer points. Here the source must be a **word** (16-bit). The source of the word can be a general purpose register, a segment register or memory.

It is important to note that whenever data is pushed onto the stack, the first (most significant) data byte moves into the stack segment memory location addressed by SP–1. The second (least significant) data byte moves into the stack segment memory location addressed by SP–2.

Examples :

1. PUSH CX ; Decrements SP by 2, copy CX to stack

The Fig. 3.12 shows the execution of PUSH CX instruction.

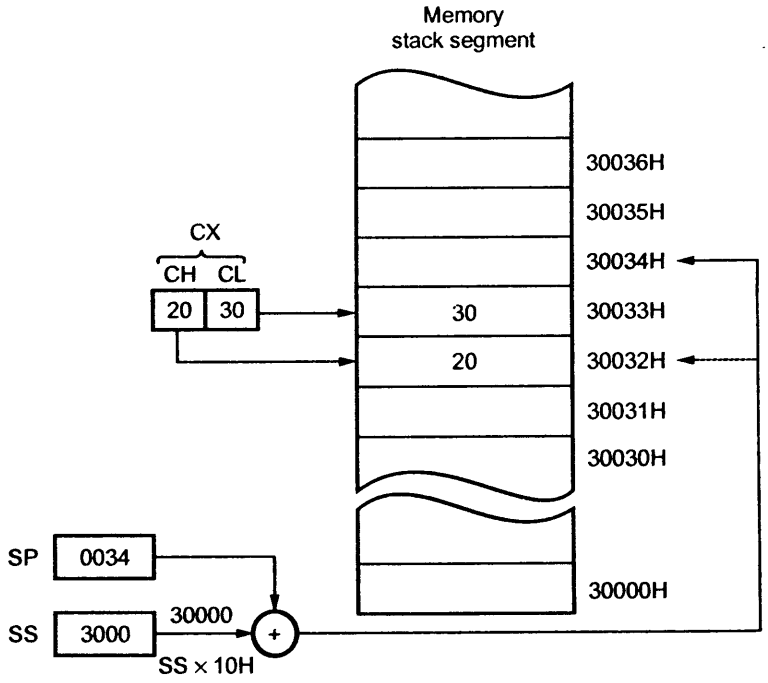


Fig. 3.12

Note : After execution of installation SP = 0032 H and it is indicated by dotted arrow.

```
2.  PUSH DS              ; Decrement SP by 2, copy DS to stack
3.  PUSH NEXT [BX]       ; Decrement SP by 2, copy a word from memory in
                         ; DS (i.e. PA = EA + DS) to stack with
                         ; EA = NEXT + [BX]
```

PUSHF

Puts the flag register contents on the stack. Whenever this instruction is executed, the most significant byte of flag register moves into the stack segment memory location addressed by SP–1. The least significant byte of flag register moves into the stack segment memory location addressed by SP–2.

POP destination

The POP instruction copies a word from the stack location pointed by the stack pointer to the destination. The destination can be a general purpose register, a segment register, or a memory location. After the word is copied to the specified destination, the stack pointer is automatically incremented by 2. Whenever data is removed from the stack, the byte from the stack segment memory location addressed by SP moves into the most significant byte of the destination register and the byte from the stack segment memory location addressed by SP + 1 moves into the least significant byte of the destination register.

Examples :

```
1.  POP CX               ; Copy a word from top of stack
                         ; to CX and increment SP by 2.
```

The Fig. 3.13 shows the execution of POP CX instruction.

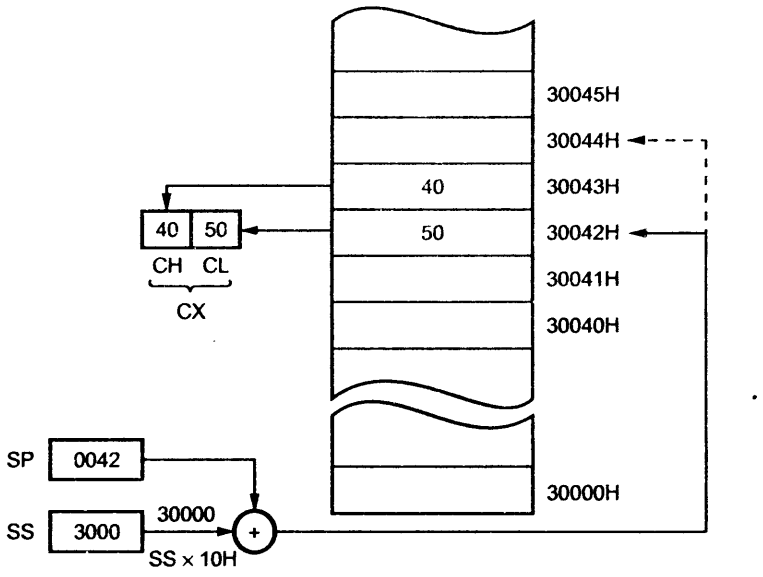


Fig. 3.13

Note : After execution of instruction SP = 0044H and it is indicated by dotted arrow.

```
2.  POP DS              ; Copy a word from top of stack
                        ; to DS and increment SP by 2.
3.  POP NEXT [BX]       ; Copy a word from top of stack to memory in DS
                        ; (i.e. PA = EA + DS) with EA = NEXT + [BX], and
                        ; increment SP by 2.
```

Note : POP CS is illegal.

POPF

Removes the word from top of stack to the flag register. Whenever this instruction is executed, the byte from the stack segment memory location addressed by SP moves into the most significant byte of the flag register and the byte from the stack segment memory location addressed by SP+1 moves into the least significant byte of the flag register.

Initializing the stack

Before going to use any instruction which uses stack for its operation we have initialize stack segment, and we have reverse the memory area required for the stack. The stack can be initialized by including following sequence of instructions in the program.

METHOD 1 :

```
ASSUME CS : CODE, DS : DATA, SS : STACK
:
STACK SEGMENT
S_DATA DB 100 DUP (?)
STACK ENDS
```

Note : Matter typed in Bold letters is included to initialize stack. This program sequence reserves 100 bytes for the stack operation.

METHOD 2 :

Syntax : · Stack [size]

Example : · Stack 100

The · stack is a directive, which provides shortcut in definition of the stack segment. The default size is 1024 bytes. The instruction stack 100 reserves 100 bytes for the stack operation.

3.4.3 Load Effective Address

The load effective address group includes following instructions,

- LEA

- LDS
- LES

LEA Instruction : Load Effective Address : LEA register, source

This instruction determines the offset of the variable or memory location named as the source and loads this address in the specified 16-bit register. Flags are not affected by LEA instruction.

Examples :

```
LEA CX, TOTAL          ; Load CX with offset of TOTAL in DS.
LEA BP, SS : STACK_TOP ; Load BP with offset of STACK_TOP in SS.
LEA AX, [BX] [DI]      ; Load AX with EA = [BX] + [DI]
```

LDS Instruction : Load register and DS with words from memory. LDS register, memory address of first word.

This instruction copies a word from two memory locations into the register specified in the instruction. It then copies a word from the next two memory locations into the DS register.

Examples :

```
LDS CX, [391AH]        ; Copy contents of memory at displacement of
                       ; 391AH and 391BH to CX. Then copy contents at
                       ; displacement of 391CH and 391DH in DS.
```

LES Instruction : Load register and ES with words from memory. LES register, memory address of first word.

This instruction loads new values into the specified register and into the ES register from four successive memory locations. The word from the first two memory location is copied into the specified register and the word from the next two memory locations is copied into the ES register.

Example :

```
LES CX, [3483H]        ; Copy contents of memory at displacement of 3483H0
                       ; in DS to CL, contents of 3484H in DS to CH and
                       ; copy the contents of memory at displacement of
                       ; 3485H and 3486H in DS to ES register.
```

3.4.4 String Data Transfer Instructions

MOVS/MOVSB/MOVSW

These instructions copy a byte or word from a location in the data segment to a location in the extra segment. The offset of the source byte or word in the data segment must be in the SI register. The offset of the destination in the extra segment must be contained in the DI register. For multiple byte or multiple word moves the number of elements to be moved is put in the CX register so that it can function as a counter. After the byte or word is moved SI and DI are automatically adjusted to point to the next source

and the next destination. If the direction flag is 0, then SI and DI will be incremented by 1 after a byte move and they will incremented by 2 after a word move. If the DF is a 1, then SI and DI will be decremented by 1 after a byte move and they will be decremented by 2 after a word move. MOVS affects no flags.

The way to tell the assembler whether to code the instruction for a byte or word move is to add a "B" or a "W" to the MOVS mnemonic. MOVSB, for example, says move a string as bytes. MOVSW says move a string as words.

Examples :

```
CLD                 ; Clear Direction Flag to autoincrement SI and DI
MOV AX, 0000H
MOV DS, AX          ; Initialize data segment register to 0
MOV ES, AX          ; Initialize extra segment register to 0
MOV SI, 2000H       ; Load offset of start of source string into SI
MOV DI, 2400H       ; Load offset of start of destination into DI
MOV CX, 04H         ; Load length of string in CX as counter
REP MOVSB           ; Decrement CX and MOVSB until CX will be 0.
```

After move SI will be one greater than offset of last byte in source string. DI will be one greater than offset of last byte of destination string. CX will be 0.

REP is a prefix which is written before MOVSB to repeat execution of it until CX = 0.

REP/REPE/REPZ/REPNE/REPNZ Prefix

REP is a prefix which is written before one of the string instructions. These instructions repeat until specified condition exists.

Instruction Code	Condition for Exit
REP	CX = 0
REPE/REPZ	CX = 0 or ZF = 0
REPNE/REPNZ	CX = 0 or ZF = 1

Examples :

```
REPZ CMP SB         ; Compare string bytes until CX = 0
                    ; or until string bytes not equal.
```

LODS/LODSB/LODSW

This instruction copies a byte from a string location pointed to by SI to AL, or a word from a string location pointed to by SI to AX. LODS does not affect any flags. LODSB copies byte and LODSW copies a word.

Examples :

```
CLD                          ; Clear direction flag so SI is autoincremented
MOV SI, OFFSET S_STRING      ; Point SI at string
LODS S_STRING.
```

STOS/STOSB/STOSW

The STOS instruction copies a byte from AL or a word from AX to a memory location in the extra segment. DI is used to hold the offset of the memory location in the extra segment. After the copy, DI is automatically incremented or decremented to point to the next string element in memory. If the direction flag, DF, is cleared, then DI will automatically be incremented by one for a byte string or incremented by two for a word string. If the direction flag is set, DI will be automatically decremented by one for a byte string or decremented by two for a word string. STOS does not affect any flags. STOSB copies byte and STOSW copies a word.

Examples :

```
MOV DI, OFFSET D_STRING      ; Point DI at destination string
STOS D_STRING                ; Assembler uses string name to determine
                             ; whether string is of type byte or type word.
                             ; If byte string, then string byte replaced
                             ; with contents of AL. If word string, then
                             ; string word replaced with contents of AX.
MOV DI, OFFSET D_STRING      ; Point DI at destination string
STOSB                        ; "B" added to STOS mnemonic directly
                             ; tells assembler to replace byte in string with byte from
                             ; AL. STOSW would tell assembler directly to replace a
                             ; word in the string with a word from AX.
```

3.4.5 Miscellaneous Data Transfer Instructions

This group consists of following instructions.

- XCHG
- LAHF
- SAHF
- XLAT
- IN and OUT

XCHG Instruction : XCHG destination, source.

The XCHG instruction exchanges the contents of a register with the contents of another register or the contents of a register with the contents of a memory location(s). The instruction cannot exchange the contents of two memory locations. The source and destination both must be words or bytes. The segment registers can't be used in these instructions.

Examples :

```
XCHG BX, CX          ; Exchange word in BX with word in CX.
XCHG AL, CL          ; Exchange byte in AL with byte in CL.
XCHG AL, SUM [BX]    ; Exchange byte in AL with byte in memory at
                     ; EA = SUM + [BX]. PA = EA + DS.
```

LAHF Instruction : Load lower byte of flag register in AH.

This instruction copies the contents of lower byte of 8086 flag register to AH register.

SAHF Instruction : Copy AH register to low byte of flag register.

The contents of the AH register are copied into the lower byte of the 8086 flag register.

XLAT / XLATB Instruction : Translate byte in AL.

The XLATB instruction replaces a byte in the AL register with a byte from a lookup table in memory. BX register stores the offset of the starting address of the lookup table and AL register stores the byte number from the lookup table. This instruction copies byte from address pointed by [BX + AL] back into AL. This instruction does not affect flags.

IN and OUT Instructions

IN Instruction : Input a byte or word from port.

The IN instruction will copy data from a port to the accumulator. If an 8-bit port is read, the data will go to AL and if an 16-bit port is read the data will goto AX.

The IN instruction can be executed in two different addressing modes,

1. Direct : In direct addressing mode 8-bit address of the port is a part of the instruction.

Examples :

```
IN AL, 0F8H          ; Copy a byte from port 0F8H to AL.
IN AX, 95H           ; Copy a word from port 95H to AX.
```

2. Indirect : In indirect addressing, the address of the port is referred from DX register. Since DX is a 16-bit register, the port address can be any number between 0000H to FFFFH. Therefore it is possible address to upto 65,536 ports in this mode.

Examples :

```
MOV DX, 30F8H        ; Load 16-bit address of the port in DX.
IN AL, DX            ;  Copy a byte from 8-bit port 30F8H to AL.
IN AX, DX            ; Copy a word from 16-bit port 30F8H to AX.
```

OUT Instruction : Send a byte or word to a port.

The OUT instruction copies a byte from AL or a word from AX to the specified port.

The OUT instruction can be executed in two different addressing modes.

1. Direct : In direct addressing mode 8-bit address of the port is a part of the instruction.

Examples :

```
OUT 0F8H, AL         ; Copy contents of AL to 8-bit port 0F8H.
OUT 0FBH, AX         ; Copy contents of AX to 16-bit port 0FBH.
```

2. Indirect : In indirect addressing, the address of the port is referred from DX register. It has advantage of accessing 2^{16} i.e. 65536 ports as mentioned earlier.

Examples :

```
MOV DX, 30F8H        ; Load 16-bit address of the port in DX.
OUT DX, AL           ; Copy the contents of AL to port 30F8H.
OUT DX, AX           ; Copy the contents of AX to port 30F8H.
```

3.5 Arithmetic and Logic Instructions

The arithmetic and logic group of instructions include

- Addition instructions
- Subtraction instructions
- Multiplication instructions
- Division
- BCD and ASCII arithmetic instructions
- Comparison
- Basic logic instructions - AND, OR NOT, XOR
- Shift and rotate instructions

3.5.1 Addition

This group of instructions consist of following instructions

- ADD : Addition
- ADC : Addition with carry
- INC : Increment (Add 1)

ADD/ADC Instruction : ADD destination, source / ADC destination, source.

These instructions add a number from source to a number from destination and put the result in the destination. The ADC, instruction also adds the status of carry flag into the result. The source may be an immediate number, a register, or a memory location. The source and the destination in an instruction cannot both be memory locations. The source and destination both must be a word or byte. If you want to add a byte to a word, you must copy the byte to a word location and fill the upper byte of the word with zeroes before adding.

Flags affected : AF, CF, OF, PF, SF, ZF.

Examples :

```
ADD AL, 0F0H          ; Add immediate number 0F0H to contents of AL.
ADC DL, CL            ; Add contents of CL to contents of DL with carry
                      ; and store result in DL i.e. DL ← DL + CL + CY
ADC DX, BX            ; Add contents of BX to contents of DX with carry
                      ; and store result in DX i.e. DX ← DX + BX + CY
ADD CL, TOTAL [BX]    ; Add byte from effective address
                      ; TOTAL [BX] to contents of CL
ADD CX, TOTAL [BX]    ; Add word from effective address
                      ; TOTAL [BX] to contents of CX.
```

INC Instruction : Increment destination.

The INC instruction adds 1 to the specified destination. The destination may be a register or memory location. The AF, OF, PF, SF and ZF flags are affected.

Examples :

```
INC AL                ; Add 1 to contents of AL.
INC BX                ; Add 1 to contents of BX.
```

NOTE : The carry flag CF is not affected.

If contents of 8-bit register are FFH and 16-bit register are FFFFH, after INC instruction contents of registers will be zero without affecting carry flag.

```
INC BYTE PTR [BX]     ; Increment byte at offset of BX in DS.
                      ; BYTE PTR directive indicates to the assembler
                      ; that the byte from memory is to be incremented.
INC WORD PTR [BX]     ; Increment word at offset of BX in DS.
                      ; WORD PTR directive indicates to the assembler
                      ; that the word from memory is to be incremented.
```

3.5.2 Subtraction

This group of instructions consist of following group of instructions.

- SUB : Subtraction
- SBB : Subtraction with borrow
- DEC : Decrement (subtract 1)
- NEG : 2's complement of a number

SUB/SBB Instruction : SUB destination, Source.

SBB destination, Source.

These instructions subtract the number in the source from the number in the destination and put result in the destination. The SBB, instruction also subtracts the status of carry flag from the result. The source may be an immediate number, a register, or a memory location. The destination may be a register or a memory location. The source and the destination both cannot be memory locations. The source and destination both must be word or byte. If you want to subtract a byte from a word, you must copy the byte to a word location and fill the upper byte of the word with zeroes before subtracting.

Flags affected : AF, CF, OF, PF, SF, and ZF.

Examples :

```
SUB AL, 0F0H          ; Subtract immediate number 0F0H
                      ; from contents of AL store result in AL.
SBB DL, CL            ; Subtract contents of CL and status of carry flag
                      ; from the contents of DL and store result in DL.
                      ; i.e. DL ← DL – CL – CY
SBB DX, BX            ; Subtract contents of BX and status of carry
                      ; flag from the DX and store result in DX.
                      ; i.e. DX ← DX – BX – CY
SUB CL, TOTAL [BX]    ; Subtract byte from effective address TOTAL [BX]
                      ; from the contents of CL and store result in CL
SUB CX, TOTAL [BX]    ; Subtract word from effective address TOTAL [BX]
                      ; from the contents of CX and store result in CX.
```

DEC Instruction : Decrement destination.

The DEC instruction subtract 1 from the specified destination. The destination may be a register or a memory location. The AF, OF, PF, SF and ZF flags are affected.

Examples :

```
DEC AL          ; Subtracts 1 from the contents of AL.
DEC BX          ; Subtracts 1 from the contents of BX.
```

Note : The carry flag CF is not affected.

If the contents of 8-bit register are 00H and 16-bit register are 0000H, after DEC instruction contents of registers will be FFH and FFFFH respectively without affecting carry flag.

```
DEC BYTE PTR [BX]   ; Decrement byte at offset of BX in DS.
                    ; BYTE PTR directive indicates to the assembler
                    ; that the byte from memory is to be decremented.
DEC WORD PTR [BX]   ; Decrement word at offset of BX in DS.
                    ; WORD PTR directive indicates to the assembler
                    ; that the word from memory is to be decremented.
```

NEG Instruction : Form 2's complement.

This instruction replaces the number in a destination with the 2's complement of that number. The destination can be a register or a memory location. This instruction can be implemented by inverting each bit and adding 1 to it.

The negate instruction updates the AF, CF, SF, PF, ZF and OF flags.

Examples :

```
                    ; AL = 0011 0101  35H
NEG AL              ; Replace number in AL with its 2's complement
                    ; AL = 1100 1011 = CBH
```

3.5.3 Comparison

The comparison instruction (CMP) compares a byte/word from the specified source with a byte/word from the specified destination. The source and destination both must be byte or word. The source may be an immediate number, a register, or a memory location. The destination may be a register or a memory location. However the source and destination both can't be memory locations. The comparison is done by subtracting the source byte or word from the destination byte or word. But the result is not stored in the destination. Source and destination remain unchanged, only flags are updated.

Flags : The AF, OF, SF, ZF, PF and CF are updated by the CMP instruction.

Examples :

```
CMP BL, 01H         ; Compare immediate number 01H with byte in BL.
CMP CX, BX          ; Compare word in BX with word in CX.
CMP CX, TOTAL       ; Compare word at displacement
                    ; TOTAL in DS with word in CX.
```

Note : It is not possible to compare segment registers.

The result of comparison is checked by conditional jump, conditional call and conditional return instructions. We discuss these instructions later in this chapter.

3.5.4 Multiplication

This group of instructions consist of following group of instructions.

- MUL : Unsigned multiplication
- iMUL : Signed multiplication

MUL Instruction : MUL source.

This instruction multiplies an unsigned byte from source and unsigned byte in AL register or unsigned word from source and unsigned word in AX register. The source can be a register or a memory location. When the byte is multiplied by the contents of AL, the result is stored in AX. The most significant byte is stored in AH and least significant byte is stored in AL. When a word is multiplied by the contents of AX, the most significant word of result is stored in DX and least significant word of result is stored in AX.

Flags : MUL instruction affect AF, PF, SF, and ZF flags.

Examples :

```
MUL BL                  ; AL × BL, result in AX.
MUL BX                  ; AX × BX, result high word in DX low word in AX.
MUL WORD PTR [BX]       ; AX times word in DS pointed by [BX]
                        ; result high word in DX low word in AX.
```

IMUL Instruction :

This instruction multiplies a signed byte from some source and a signed byte in AL, or a signed word from some source and a signed word in AX. The source can be register or memory location. When a signed byte is multiplied by AL a signed result will be put in AX. When a signed word is multiplied by AX, the high-order word of the signed result is put in DX and the low-order word of the signed result is put in AX.

If the upper byte of a 16-bit result or the upper word of 32-bit result contains only copies of the sign bit (all 0's or all 1's), then the CF and the OF flags will both be 0's. The AF, PF, SF, and ZF flags are undefined after IMUL.

To multiply a signed **byte** by a signed **word** it is necessary to move the byte into a word location and fill the upper byte of the word with copies of the sign bit. This can be done using CBW instruction.

Examples :

```
IMUL BL                 ; AL × BL, result in AX
IMUL CX                 ; AX × CX, high-order word of result in DX and
                        ; low-order word of result in AX.
```

3.5.5 Division

This group of instructions consists of following group of instructions

- DIV
- IDIV.

DIV Instruction : DIV source

This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word by a word.

When dividing a word by a byte, the word must be in AX register. After the division AL will contain an 8-bit quotient and AH will contain an 8-bit remainder. If an attempt is made to divide by 0 or the quotient is too large to fit in AL (greater than FFH), the 8086 will automatically execute a type 0 interrupt.

When a double word is divided by a word, the most significant word of the double word must be in DX and the least-significant word must be in AX. After the division AX will contain a 16-bit quotient and DX will contain a 16-bit remainder. Again, if an attempt is made to divide by 0 or the quotient is too large to fit in AX register (greater than FFFFH), the 8086 will do a type 0 interrupt. For DIV instruction source may be a register or memory location.

To divide a byte by a byte, it is necessary to put the dividend byte in AL and fill AH with all 0's. Similarly, to divide a word by a word, it is necessary to put the dividend word in AX and fill DX with all 0's.

Flags : All flags are undefined after a DIV instruction.

Examples :

```
DIV CL      ; Word in AX/byte in CL,
            ; Quotient in AL, remainder in AH.
DIV CX      ; Double word in DX and AX/word in CX,
            ; Quotient in AX, remainder in DX.
```

IDIV Instruction : IDIV source

This instruction is used to divide a signed word by a signed byte, or to divide a signed double word (32-bits) by a signed word. Rest all is similar to DIV instruction.

3.5.6 BCD and ASCII Arithmetic

The 8086 allows arithmetic manipulation of both BCD (Binary Coded Decimal) and ASCII (American Standard Code for Information Interchange) data. This is accomplished by instructions that adjust the numbers for BCD and ASCII arithmetic. Let us see instructions used for BCD and ASCII arithmetic.

3.5.6.1 BCD Arithmetic

The 8086 provides two instructions to support BCD arithmetic. They correct result of a BCD addition and a BCD subtraction. The DAA (decimal adjust after addition) instruction follows BCD addition, and the DAS (Decimal Adjust After Subtraction) follows BCD subtraction. Both instructions correct the result of the addition or subtraction so that it is a BCD number.

DAA Instruction : Decimal Adjust Accumulator.

This instruction is used to make sure the result of adding two packed BCD numbers is adjusted to be a legal BCD number.

Instruction works as follows :

1. If the value of the low-order four bits (D_3-D_0) in the AL is greater than 9 or if AF is set, the instruction adds 6 (06) to the low-order four bits.
2. If the value of the high-order four bits (D_7-D_4) in the AL is greater than 9 or if carry flag is set, the instruction adds 6 (60) to the high-order four bits.

Examples :

```
1.                  ; AL = 0011 1001 = 39 BCD
                    ; CL = 0001 0010 = 12 BCD
Add AL, CL          ; AL = 0100 1011 = 4BH
DAA                 ; Add 0110 Because 1011 > 9
                    ; AL = 0101 0001 = 51 BCD
2.                  ; AL = 1001 0110 = 96 BCD
                    ; BL = 0000 0111 = 07 BCD
ADD AL, BL          ; AL = 1001 1101 = 9DH
DAA                 ; Add 0110 Because 1101 > 9
                    ; AL = 1010 0011 = A3H
                    ; 1010 > 9 so add 0110 0000
                    ; AL = 0000 0011 = 03 BCD, CF = 1. The result is 103.
```

The instruction updates the AF, CF, PF, and ZF. The OF is undefined after DAA instruction.

Note : Only works for AL.

DAS Instruction : Decimal Adjust After Subtraction.

This instruction is used after subtracting two packed BCD numbers to make sure the result is correct packed BCD. Instruction works as follows :

1. If the value of the low-order four bits (D_3-D_0) in the AL is greater than 9 or if AF is set; the instruction subtracts 6 (06) from the low-order four bits.
2. If the value of the high-order four bits (D_7-D_4) in the AL is greater than 9 or if carry flag is set, the instruction subtracts 6 (60) from the high-order four bits.

Examples :

```
1.                  ; AL = 0011 0010 = 32 BCD
                    ; CL = 0001 0111 = 17 BCD
SUB AL, CL          ; AL = 0001 1011 = 1BH
                    ; Subtract 0110 Because 1011 > 9
                    ; AL = 0001 0101 = 15 BCD
2.                  ; AL = 0010 0011 = 23 BCD
                    ; CL = 0101 1000 = 58 BCD
SUB AL, CL          ; AL = 1100 1011 =  CBH CF = 1
                    ; Subtract 0110 (6) Because 1011 > 9
                    ; AL = 1100 0101 = C5H
                    ; Subtract 0110 0000 Because 1100 > 9
                    ; AL = 0110 0101 = 65 BCD  CF = 1,
                    ; CF = 1 means  borrow
                    ; is needed means number is negative (– 65).
```

The DAS instruction updates the AF, CF, PF, and ZF. The OF flag is undefined after DAS instruction.

Note : DAS only works for AL.

3.5.6.2 ASCII Arithmetic

ASCII numbers range in value from 30H to 39H for the numbers 0-9. The 8086 provides four instructions for ASCII arithmetic.

- AAA:ASCII adjust after addition
- AAS :ASCII adjust after subtraction
- AAM:ASCII adjust after multiplication
- AAD :ASCII adjust before division

AAA Instruction : ASCII Adjust for Addition.

The numbers from 0-9 are represented as 30H-39H in ASCII code. When you want to add two decimal digits which are represented in ASCII code, it is necessary to mask upper nibble (3) from the code before addition. The 8086 allows you to add the ASCII codes for two decimal digits without masking off the "3" in the upper nibble of each digit. The AAA instruction can be used after addition to get the current result in unpacked BCD form.

Examples :

```
                    ; AL   = 0011 0100 ASCII 4
                    ; CL   = 0011 1000 ASCII 8
ADD AL, CL          ; AL   = 0110 1100
                    ; 6CH  = Incorrect temporary result
AAA                 ; AL   = 0000 0010 Unpacked BCD for 2.
```

; Carry = 1 to indicate correct answer is 12 decimal.

The AAA instruction updates the AF and the CF, but the OF, PF, SF, and ZF are left undefined.

Note : The AAA instruction only works on the AL register.

AAS Instruction : ASCII Adjust After Subtraction.

The numbers from 0-9 are represented as 30-39 in ASCII code. When you want to subtract two decimal digits which are represented in ASCII code, it is necessary to mask upper nibble (3) from the code before subtraction. The 8086 allows you to subtract the ASCII codes for two decimal digits without masking off the "3" in the upper nibble of each digit. The AAS instruction can be used after subtraction to get the current result in unpacked BCD form.

Examples :

```
1.                  ; AL = 0011 1000 ASCII 8
                    ; CL = 0011 0010 ASCII 2
SUB AL, CL          ; AL = 0000 0110 BCD 06
                    ; CF = 0
AAS                 ; AL = 0000 0010 = BCD 06
                    ; CF = 0 no borrow required
2.                  ; AL = 0011 0010 ASCII 2
                    ; CL = 0011 1000 ASCII 8
SUB AL, CL          ; AL = 1111 1010 = FAH
                    ; CF = 1
AAS                 ; AL = 0000 0110 = BCD 6
                    ; CF = 1 borrow needed means (– 6)
```

AAM Instruction : ASCII Adjust After Multiplication.

After the two unpacked BCD digits are multiplied, the AAM instruction is used to adjust the product to two unpacked BCD digits in AX.

Examples :

```
                    ; AL = 0000 0100 = Unpacked BCD 4
                    ; CL = 0000 0110 = Unpacked BCD 6
MUL CL              ; AL × CL Result in AX.
                    ; AX = 0000 0000 0001 1000 = 0018H
AAM                 ; AX = 0000 0010 0000 0100 = 0204H
                    ; Which is unpacked BCD for 24.
```

Now by adding 3030H in AX register we get the result in ASCII form.

AAD Instruction : ASCII Adjust Before Division.

AAD converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. After the division AL will contain the unpacked BCD quotient and AH will contain the unpacked BCD remainder. The PF, SF and ZF are updated. The AF, CF and OF are undefined after AAD.

Examples :

```
                    ; AX = 0403 unpacked BCD for 43 decimal, CL = 07H
AAD                 ; Adjust to binary before division,
                    ; AX = 002BH = 2BH = 43 decimal.
DIV CL              ; Divide AX by unpacked BCD in CL.
                    ; AL = quotient = 06 unpacked BCD
                    ; AH = remainder = 01 unpacked BCD
```

Now by adding 3030H in AX register we get the quotient and remainder in ASCII form.

3.5.7 Basic Logic Instructions

The basic logic instructions include AND, OR, Exclusive-OR, and NOT. This group also includes TEST instruction which is a special form of the AND instruction.

AND Instruction : AND destination, source.

We know that, AND operation with two inputs produces result logic 1 only when both the inputs are logic 1. i.e. $Y = A \cdot B$.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.2 Truth table for AND gate

This instruction logically ANDs each bit of the source byte or word with the corresponding bit in the destination and stores result in the destination. The source may be an immediate number, a register or a memory location. The destination may be a register

or a memory location. The source and destination both cannot be memory locations in the same instruction. The CF and OF are both 0 after AND. The PF, SF and ZF are affected. AF is undefined.

Examples :

```
1.                  ; AL = 1001 0011 = 93H
                    ; BL = 0111 0101 = 75H
AND BL, AL          ; AND byte in AL with byte in BL
                    ; BL = 0001 0001 = 11H
2.                  ; CX = 0110 1011 1001 1110
AND CX, 00F0H       ; CX = 0000 0000 1001 0000
```

The AND operation clears bits of a binary number. The task of clearing a bit in a binary number is called **masking.** The Fig. 3.14 shows the process of masking.

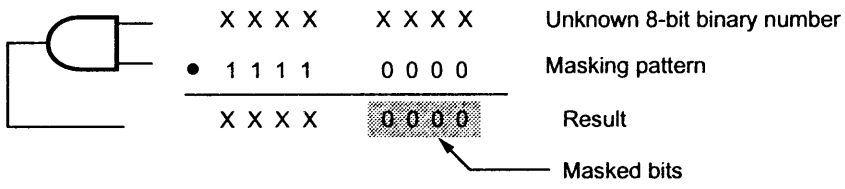


Fig. 3.14 Masking using AND operation

OR Instruction : OR destination, source.

We know that, OR operation with two inputs produces result logic 1 when any one or both inputs are logic 1 i.e., Y = A + B.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.3 Truth table for OR gate

This instruction logically ORs each bit of the source byte or word with the corresponding bit in the destination and stores result in the destination. The source may be an immediate number, a register or a memory location. The destination may be a register

or a memory location. The source and destination both cannot be memory locations in the same instruction. The CF and OF are both 0 after OR. The PF, SF and ZF are affected. AF is undefined.

Examples :

```
1.                  ; AL = 1001 0011 = 93H
                    ; BL = 0111 0101 = 75H
OR BL, AL           ; OR byte in AL with byte in BL.
                    ; BL = 1111 0111 = F7H
2.                  ; CX = 0110 1011 1001 1110
OR CX, 00F0H        ; CX = 0110 1011 1111 1110
```

The OR instruction is used to set (make one) any bit in the binary number. This is illustrated in Fig. 3.15.

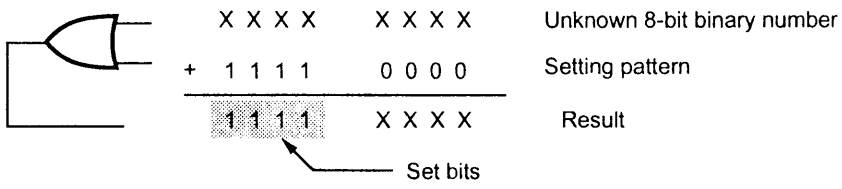


Fig. 3.15 Setting bit/s using OR operation

XOR Instruction : XOR destination, source.

We know that, XOR operation produces result logic 1 when odd number of inputs are logic 1 i.e. $Y = A\overline{B} + \overline{A}B$.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.4 Truth table for XOR gate

This instruction logically XORs each bit of the source byte or word with the corresponding bit in the destination and stores result in the destination. The source may be

an immediate number, a register or a memory location. The destination may be a register or a memory location. The source and destination both cannot be memory locations in the same instruction. The CF and OF are both 0 after XOR. The PF, SF and ZF are affected. AF is undefined.

Examples :

```
1.              ; AL = 1010  1111 = AFH
                ; BL =  1111  0000 = F0H
XOR BL, AL      ; XOR byte in AL with byte in BL
                ; BL =  0101  1111 = 5FH
```

The XOR instruction is used if some bits of a register or memory location must be inverted. This instruction allows part of a number to be inverted or complemented. This is illustrated in Fig. 3.16.

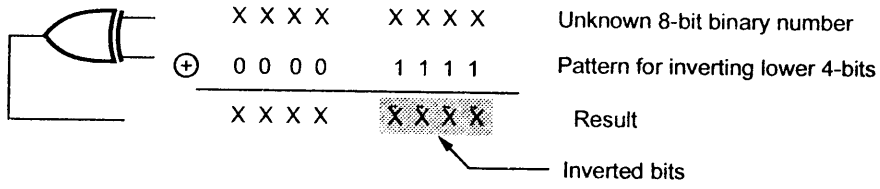


Fig. 3.16 Inversion of part of a number using XOR operation

NOT Instruction : NOT destination.

The NOT instruction inverts each bit of a byte or a word. The destination can be register or a memory location.

Flags : NOT instruction affects no flag.

Examples :

```
                ; AL = 0110 1100
NOT AL          ; AL = 1001 0011
                ; CX = 1010 1111 0010 0110
NOT CX          ; CX = 0101 0000 1101 1001
```

Test and bit test Instructions :

The TEST instruction performs the AND operation. The difference is that the AND instruction changes the destination operand, while the TEST instruction does not. A TEST only affects the condition of the flag register, which indicates the result of the test.

PF, SF and ZF will be updated to show the results of the ANDing. PF has meaning only for the lower 8-bits of the destination. AF will be undefined.

Examples :

```
TEST AL, CL        ; AND CL with AL.
                   ; Update flags, result is not stored.
TEST BX, CX        ; AND CX with BX.
                   ; Update flags, result is not stored.
```

The TEST instruction functions in the similar manner as a CMP instruction. The difference is that the TEST instruction normally tests a single bit (or occasionally multiple bits), while the CMP instruction tests the entire byte or word. The Fig. 3.17 shows the bit pattern and test operation for testing of bit 0. If zero flag is set (Z = 1) after this operation, the bit under test bit-0 is zero ; otherwise bit-0 is 1.

The zero flag is usually tested by JZ or JNZ instructions. Therefore, the TEST instruction is usually followed by either the JZ or JNZ instruction.

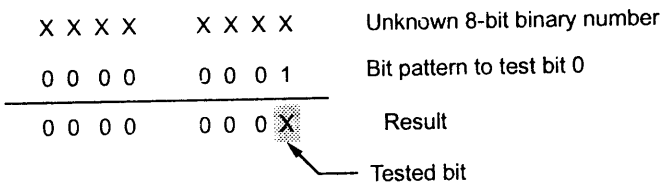


Fig. 3.17 TEST operation

3.5.8 Shift and Rotate

3.5.8.1 Shift

Shift instructions position or move binary data to the left or right by shifting them within the register or memory location. They also perform multiplication by powers of 2^{+n} (left shift) and division by powers of 2^{-n} (right shift). The shift operations can be classified as logical shifts and arithmetic shifts. The logical shifts move a 0 into the rightmost bit position for a logical left shift (SHL) and a 0 into the leftmost bit position for a logical right shift (SHR). The arithmetic left shift (SAL) and logical left shift operations are identical. However, arithmetic and logical right shifts are different because the arithmetic right shift (SAR) copies the sign bit through the number, while the logical right shift copies a 0 through the number. This is illustrated in Fig. 3.18. Logical shift operations are used with unsigned numbers; they perform multiplication or division of unsigned numbers. On the otherhand, arithmetic shift operations are used with signed numbers; they perform multiplications or division of signed numbers.

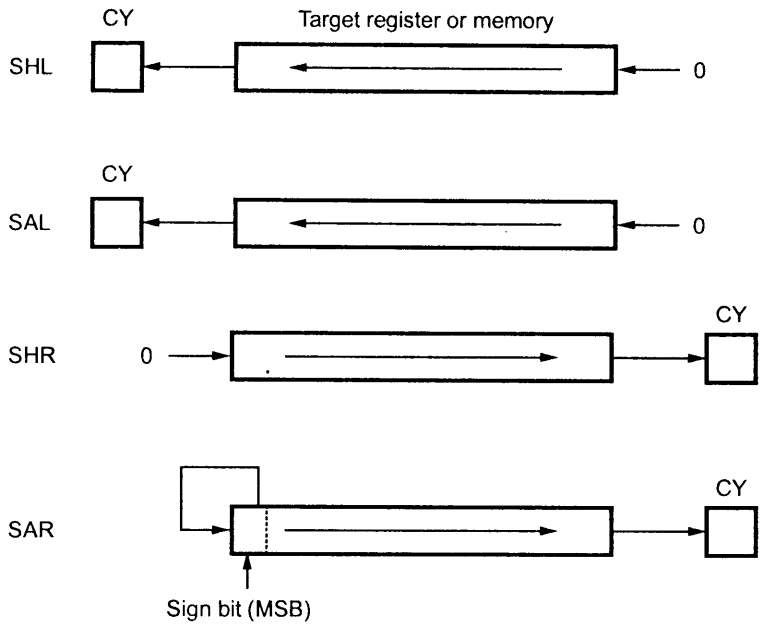
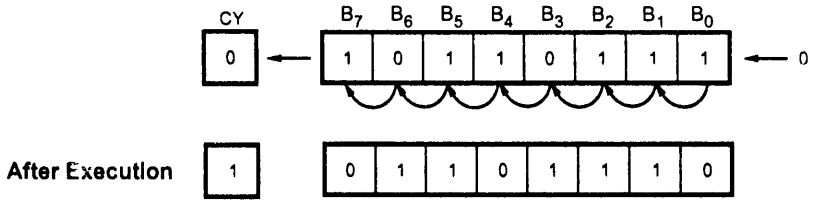


Fig. 3.18 Shift operations

SAL/SHL Instruction : SAL/SHL destination, count.

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination to the left and 0 is stored at LSB position. The MSB is shifted into the carry flag. The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. But if the number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position of the instruction.

Diagram shows SAL instruction for byte operation.



Flags : All flags are affected.

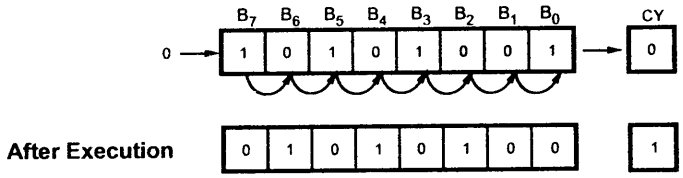
Examples :

```
SAL CX, 1          ; Shift word in CX 1 bit position left, 0 in LSB
MOV CL, 05H        ; Load desired number of shifts in CL
SAL AX, CL         ; Shift word in AX left 5 times
                   ; 0s in 5 least-significant bits.
```

SHR Instruction : SHR destination, count

This instruction shifts each bit in the specified destination to the right and 0 is stored at MSB position. The LSB is shifted into the carry flag. The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. If number of shifts required is one, you can place 1 in the count position. But if the number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position of the instruction.

Diagram shows SHR instruction for byte operation.



Flags : All flags are affected.

Examples :

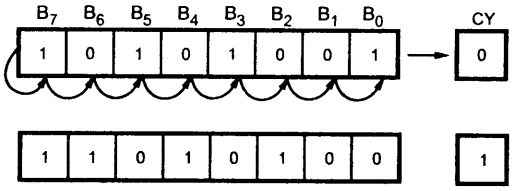
```
SHR CX, 1          ; Shift word in CX 1 bit position right, 0 in MSB.
MOV CL, 05H        ; Load desired number of shifts in CL.
SHR AX, CL         ; Shift word in AX right 5 times
                   ; 0's in 5 most significant bits.
```

SAR Instruction : SAR destination, count.

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. The LSB will be shifted into CF. In the case of multiple shifts, CF will contain the bit most recently shifted in from the LSB. Bits shifted into CF previously will be lost.

The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. If number of shifts required is one, you can place 1 in the count position. If number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position of the instruction.

Diagram shows SAR instruction for byte operation.



Flags : All flags are affected.

Examples :

```
SAR BL, 1          ; Shift byte in BL one bit position right.
MOV CL, 04H        ; Load desired number of shifts in CL.
SAR DX, CL         ; Shift word stored in DX 4-bit positions right.
```

3.5.8.2 Rotate

Rotate instructions position or move binary data by rotating the information in a register or memory location, either from one end to another or through the carry flag. This is illustrated in Fig. 3.19.

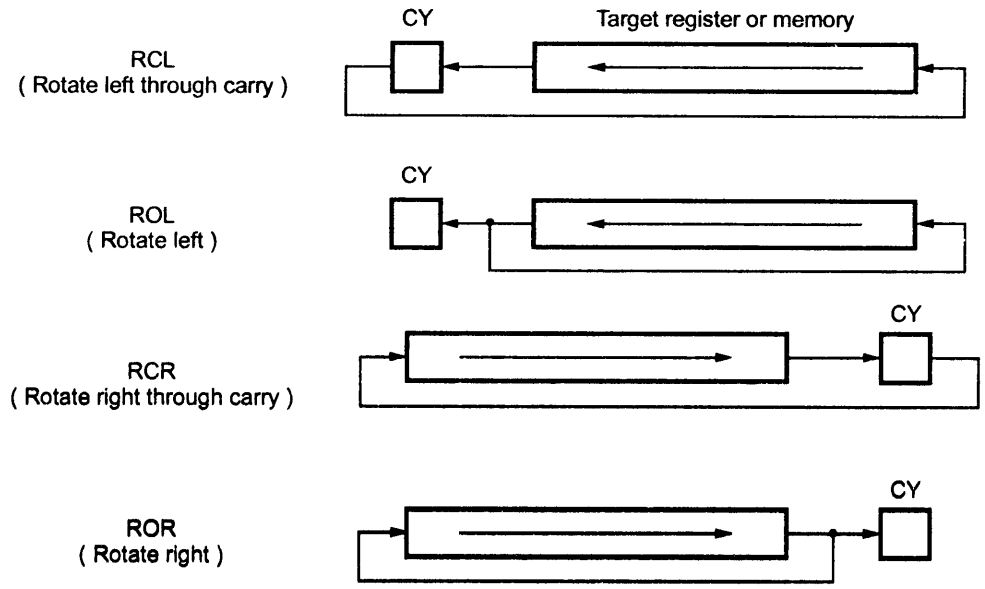
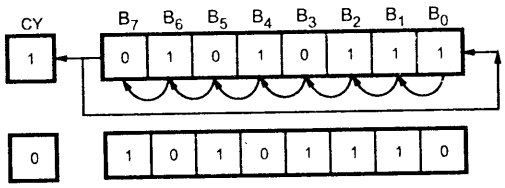


Fig. 3.19 Rotate operations

ROL Instruction : ROL destination, count.

This instruction rotates all bits in a specified byte or word to the left some number of bit positions. MSB is placed as a new LSB and a new CF.

Diagram shows ROL instruction for byte rotation.



The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. If number of shifts required is one you can place 1 in the count position. If number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position of the instruction.

Examples :

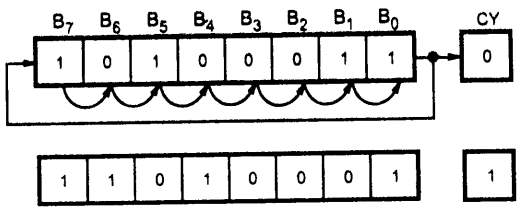
```
ROL CX, 1      ; Word in CX one bit position left, MSB to LSB and CF
MOV CL, 03H    ; Load desired number of bits to rotate in CL.
ROL BL, CL     ; Rotate BL three positions.
```

ROR Instruction : ROR destination, count.

This instruction rotates all bits in a specified byte or word to the right some number of bit positions. LSB is placed as a new MSB and a new CF.

The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. If number of shifts required is one, you can place 1 in the count position. If number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position of the instruction.

Diagram shows ROR instruction for byte rotation.



Examples :

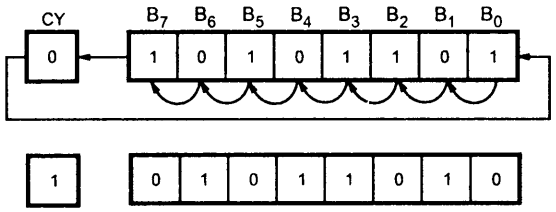
```
ROR CX, 1          ; Rotated word in CX one bit position right,
                   ; LSB to MSB and CF.
MOV CL, 03H        ; Load number of bits to rotate in CL.
ROR BL, CL         ; Rotate BL three positions.
```

RCL Instruction : RCL destination, count.

This instruction rotates all of the bits in a specified word or byte some number of bit positions to the left along with the carry flag. MSB is placed as a new carry and previous carry is placed as a new LSB.

The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. If number of shifts required is one, you can place 1 in the count position. If number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position of the instruction.

Diagram shows RCL instruction for byte rotation.



Examples :

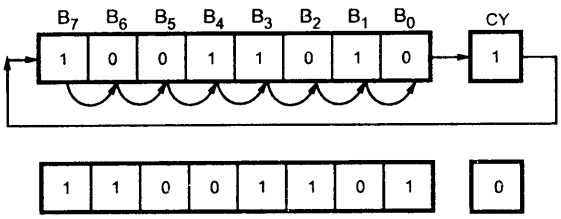
```
RCL CX, 1          ; Rotated word in CX 1-bit left, MSB to CF, CF to LSB.
MOV CL, 04H        ; Load number of bit positions to rotate in CL.
RCL AL, CL         ; Rotate AL 4-bits left.
```

RCR Instruction : RCR destination, count.

This instruction rotates all of the bits in a specified word or byte some number of bit positions to the right along with the carry flag. LSB is placed as a new carry and previous carry is placed as a new MSB.

The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts are indicated by count. If number of shifts required is one you can place 1 in the count position. If number of shifts are greater than 1 then shift count must be loaded in CL register and CL must be placed in the count position in the instruction.

Diagram shows RCR instruction for byte rotation.



Examples :

```
RCR CX, 1          ; Word in CX 1-bit right, LSB to CF, CF to MSB.
MOV CL, 04H        ; Load number of bit positions to rotate in CL.
RCR AL, CL         ; Rotate AL 4 bits right.
```

3.6 String Instructions

The 8086 instruction set provides following string instructions.

- REP/REPE/REPZ/REPNE/REPNZ
- MOVS/MOVSB/MOVSW
- LODS/LODSB/LODSW
- STOS/STOSB/STOSW
- CMPS/CMPSB/CMPSW
- SCAS/SCASB/SCASW

From the above six instructions we have already studied first four instructions in section 3.4. the remaining two instructions are string compare instructions. The string comparison instructions allow the programmer to test a section of memory against a constant or against another section of memory.

CMPS/CMPSB/CMPSW Instruction :

A string is a series of the same type of data items in sequential memory locations. The CMPS instruction can be used to compare a byte in one string with a byte in another string or to compare a word in one string with a word in another string. SI is used to hold the offset of a byte or word in the source string and DI is used to hold the offset of a byte or a word in the other string. The comparison is done by subtracting the byte or word pointed to by DI from the byte or word pointed to by SI. The AF, CF, OF, PF, SF, and ZF flags are affected by the comparison, but neither operand is affected.

Examples :

```
                          ; Point SI at source string, Point DI at
                          ; destination string
MOV SI, OFFSET F_STRING
MOV DI, OFFSET S_STRING
CLD                       ; DF cleared so SI and DI will
                          ; autoincrement after compare
CMPS F_STRING, S_STRING   ; The assembler uses names to determine whether
                          ; strings were declared as type byte or as type
                          ; word.
MOV CX, 100               ; Put number of string elements in CX, Point SI at
                          ; source of string and DI at destination of string
MOV SI, OFFSET F_STRING
MOV DI, OFFSET S_STRING
STD                       ; DF set so SI and DI will autodecrement after
                          ; compare
REPE CMPSB                ; Repeat the comparison of string bytes until end
                          ; of string or until compared bytes are not equal.
```

After the comparison SI and DI will be automatically incremented or decremented according to direction flag to point to the next element in the two strings (if DF = 0, SI and DI ↑ otherwise ↓) CX functions as a counter which is decremented after each comparison. This will go on until CX = 0.

SCAS/SCASB/SCASW Instruction :

SCAS compares a string byte with a byte in AL or a string word with word in AX. The instruction affects the flags, but it does not change either the operand in AL (AX) or the operand in the string. The string to be scanned must be in the extra segment and DI must contain the offset of the byte or the word to be compared.

After the comparison DI will be automatically incremented or decremented according to direction flag to point to the next element in the two strings (if DF = 0, SI and DI ↑ otherwise ↓) CX functions as a counter which is decremented after each comparison. This will go on until CX = 0. SCAS affects the AF, CF, OF, PF, SF and ZF flags.

Examples :

```
                          ; Scan a text string of 80 characters
                          ; for a carriage return
MOV AL, 0DH               ; Byte to be scanned for into AL
MOV DI, OFFSET TEXT_STRING ; Offset of string to DI
MOV CX, 80                ; CX used as element counter
CLD                       ; Clear DF, so DI autoincrements
```

```
REPNE SCAS TEXT_STRING          ; Compare byte in string with byte in
                                ; AL.
```

SCASB says compare strings as bytes and SCASW says compare strings as words.

3.7 Program Control Transfer Instructions

These instructions are classified as :

- Unconditional transfer instructions - CALL, RET, JMP
- Conditional transfer instructions - J cond.

3.7.1 CALL and RET Instructions

Whenever we need to use a group of instructions several times throughout a program there are two ways we can avoid having to write the group of instructions each time we want to use them. One way is to write the group of instructions as a separate procedure. We can then just CALL the procedure whenever we need to execute that group of instructions. For calling the procedure we have to store the return address onto the stack. This process takes some time. If the group of instructions is big enough then this overhead time is negligible with respect to execution time. But if the group of instructions is too short, the overhead time and execution time are comparable. In such cases, it is not desirable to write procedures. For these cases, we can use macros. Macro is also a group of instructions. Each time we "CALL" a macro in our program, the assembler will insert the defined group of instructions in place of the "CALL". An important point here is that the assembler generates machine codes for the group of instructions each time macro is called. So there is not overhead time involved in calling and returning from a procedure. The disadvantage of macro is that it generates inline code each time when the macro is called which takes more memory. In this section we discuss the procedures.

From the above discussions, we know that the procedure is a group of instructions stored as a separate program in the memory and it is called from the main program whenever required. The type of procedure depends on where the procedure is stored in the memory. If it is in the same code segment where the main program is stored then it is called **near procedure** otherwise it is referred to as **far procedure**. For near procedure CALL instruction pushes only the IP register contents on the stack, since CS register contents remains unchanged for main program and procedure. But for far procedures CALL instruction pushes both IP and CS on the stack. Let us see the detail description and examples of CALL instruction to enter the procedure and RET instruction to return from the procedure.

CALL Instruction :

The CALL instruction is used to transfer execution to a subprogram or procedure. There are two basic types of CALLs, near and far. A **near CALL** is a call to a procedure which is in the same code segment as the CALL instruction. When the 8086 executes a near CALL instruction it decrements the stack pointer by two and copies the offset of the next instruction after the CALL on the stack. It loads IP with the offset of the first instruction of the procedure in same segment.

A **far CALL** is a call to a procedure which is in a different segment from that which contains the CALL instruction. When the 8086 executes a far CALL it decrements the stack pointer by two and copies the contents of the CS register to the stack. It then decrements the stack pointer by two again and copies the offset of the instruction after the CALL to the stack. Finally, it loads CS with the segment base of the segment which contains the procedure and IP with the offset of the first instruction of the procedure in that segment.

Examples :

Direct within segment (near)

```
CALL PRO          ; PRO is the name of the procedure.
                  ; The assembler determines displacement of pro
                  ; from the instruction after the CALL and codes
                  ; this displacement in as part of the instruction.
```

Indirect within-segment (near)

```
CALL CX           ; CX contains, the offset of the first instruction
                  ; of the procedure. Replaces contents of IP with
                  ; contents of register CX.
```

Indirect to another segment (far)

```
CALL DWORD PTR [BX]  ; New values for CS and IP are fetched from four
                     ; memory locations in DS. The new value for CS
                     ; is fetched from [BX] and [BX + 1], the new IP
                     ; is fetched from [BX + 2] and [BX + 3].
```

RET Instruction :

The RET instruction will return execution from a procedure to the next instruction after the CALL instruction in the calling program. If the procedure is a near procedure (in the same code segment as the CALL instruction), then the return will be done by replacing the instruction pointer with a word from the top of the stack.

If the procedure is a far procedure (in a different code segment from the CALL instruction which calls it), then the instruction pointer will be replaced by the word at the top of the stack. The stack pointer will then be incremented by two. The code segment register is then replaced with a word from the new top of the stack. After the code segment word is popped off the stack, the stack pointer is again incremented by two. These words/word are the offset of the next instruction after the CALL. So 8086 will fetch the next instruction after the CALL.

As explained earlier, a near type jump instruction can cause the next instruction to be fetched from anywhere in the current code segment. To produce the new instruction fetch address, this instruction adds a 16-bit signed displacement contained in the instruction to the contents of the instruction pointer register. A 16-bit signed displacement means that the jump can be to a location anywhere from +32,767 to –32,768 bytes from the current instruction pointer location. A positive displacement usually means jump is "ahead" in the program, and a negative displacement usually means jump is "backward" in the program.

A special case of the direct near jump instruction is direct short jump. If the destination for the jump is within a displacement range of +127 to –128 bytes from the current instruction pointer location, the destination can be reached with just an 8-bit displacement.

3.7.3 Cond - Conditional Jump

Conditional jumps are always short jumps in the 8086. These instructions will cause a jump to a label given in the instruction if the desired condition(s) occurs in the program before the execution of the instruction. The destination must be in the range of –128 bytes to +127 bytes from the address of the instruction after the conditional transfer instruction. If the jump is not taken, execution simply goes on to the next instruction.

Instruction Code	Description	Condition for jump
JA/JNBE	Jump if above/Jump if not below or equal.	CF = 0 and ZF = 0
JAE/JNB	Jump if above or equal/Jump if not below.	CF = 0 and ZF = 1
JB/JNAE/JC	Jump if below/Jump if not above or equal.	CF = 1 and ZF = 0
JBE/JNA	Jump if below or equal/Jump if not above.	CF = 1 and ZF = 1
JE/JZ	Jump if equal/Jump if zero flag.	ZF = 1
JG/JNLE	Jump if greater/Jump if not less than nor equal.	ZF = 0 and CF = 0
JGE/JNL	Jump if greater than or equal/Jump if not less than.	SF = 0
JL/JNGE	Jump if less than/Jump if not greater than or equal.	SF ≠ 0
JLE/JNG	Jump if less than or equal/Jump if not greater	ZF = 1 or SF ≠ 0
JNC	Jump if no carry	CF = 0
JNE/JNZ	Jump if not equal/Jump if not zero	ZF = 0
JNO	Jump if no overflow	OF = 0
JNP/JPO	Jump if not parity/Jump if parity odd	PF = 0
JNS	Jump if not sign or jump if positive	SF = 0
JO	Jump if overflow flag = 1.	OF = 1
JP/JPE	Jump if parity/Jump if parity even	PF = 1
JS	Jump if sign flag = 1 or jump if negative	SF = 1
JCXZ	Jump if CX is zero	CX = 0

Note : The terms greater and less are used to refer to the relationship of two signed numbers.

3.8 Iteration Control Instructions

These instructions are used to execute a series of instructions some number of times. The number is specified in the CX register. The CX register is automatically decremented by one, each time after execution of LOOP instruction. Until CX = 0, execution will jump to a destination specified by a label in the instruction.

The destination address for the jump must be in the range of – 128 bytes to + 127 bytes from the address of the instruction after the iteration control instruction. For LOOPE/LOOPZ and LOOPNE/LOOPNZ instructions there is one more condition for exit from loop, which is given below. If the loop is not taken, execution simply goes on to the next instruction after the iteration control instruction.

Instruction Code	Description	Condition for Exit
1. LOOP	Loop through a sequence of instructions	CX = 0
2. LOOPE/LOOPZ	Loop through a sequence of instructions	CX = 0 or ZF = 0
3. LOOPNE/LOOPNZ	Loop through a sequence of instructions	CX = 0 or ZF = 1

3.9 Processor Control Instructions

- STC
- CLC
- CMC
- STD
- CLD
- STI
- CLI

STC Instruction :

This instruction sets the carry flag, STC does not affect any other flag.

CLC Instruction :

This instruction resets the carry flag to zero. CLC does not affect any other flag.

CMC Instruction :

This instruction complements the carry flag. CMC does not affect any other flag.

STD Instruction :

This instruction is used to set the direction flag to one so that SI and/or DI can be decremented automatically after execution of string instructions. STD does not affect any other flag.

CLD Instruction :

This instruction is used to reset the direction flag to zero, so that SI and/or DI can be incremented automatically after execution of string instructions. CLD does not affect any other flag.

STI Instruction :

This instruction sets the interrupt flag to one. This enables INTR interrupt of the 8086. STI does not affect any other flag.

CLI Instruction :

This instruction resets the interrupt flag to zero. Due to this 8086 will not respond to an interrupt signal on its INTR input. CLI does not affect any other flag.

3.10 External Hardware Synchronization Instructions

- HLT
- WAIT
- ESC
- LOCK
- NOP

HLT Instruction :

The HLT instruction will cause the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state. The only ways to get the processor out of the halt state are with an interrupt signal on the INTR pin, an interrupt signal on the NMI pin, or a reset signal on the RESET input.

WAIT Instruction :

When this instruction executes, the 8086 enters an idle condition where it is doing no processing. The 8086 will stay in this idle state until a signal is asserted on the 8086 TEST input pin, or until a valid interrupt signal is received on the INTR or the NMI interrupt input pins. If a valid interrupt occurs while the 8086 is in this idle state, the 8086 will return to the idle state after the execution of interrupt service procedure. WAIT affects no flags. The WAIT instruction is used to synchronize the 8086 with external hardware such as the 8087 math coprocessor.

ESC Instruction :

This instruction is used to pass instructions to a coprocessor such as the 8087 math coprocessor which shares the address and data bus with an 8086. Instructions for the coprocessor are represented by a 6-bit code embedded in the escape instruction. When the 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the

action specified by the 6-bit code specified in the instruction. In most cases the 8086 treats the ESC instruction as a NOP. In some cases the 8086 will access a data item in memory for the coprocessor.

LOCK Instruction :

In a multiprocessor system each microprocessor has its own local buses and memory. The individual microprocessors are connected together by a system bus so that each can access system resources such as disk drives or memory. Each microprocessor only takes control of the system bus when it needs to access some system resources. The LOCK prefix allows a microprocessor to make sure that another processor does not take control of the system bus while it is in the middle of a critical instruction which uses the system bus. The LOCK prefix is put in front of the critical instruction. When an instruction with a LOCK prefix executes, the 8086 will assert its bus lock signal output. This signal is connected to an external bus controller device which then prevents any other processor from taking over the system bus. LOCK affects no flags.

Examples :

```
LOCK XCHG SEMAPHORE, AL  ; The XCHG instruction requires two bus
                         ; accesses.
                         ; The LOCK prefix prevents another processor
                         ; from taking control of the system bus between
                         ; the two accesses.
```

NOP Instruction :

At the time of execution of NOP instruction, no operation is performed except fetch and decode. It takes three clock cycles to execute the instruction. NOP instruction does not affect any flag. This instruction is used to fill in time delays or to delete and insert instructions in the program while trouble shooting.

3.11 Interrupt Instructions

INT Instruction : INT Type

This instruction causes the 8086 to call a far procedure. The term type in the instruction refers to a number between 0-255 which identifies the interrupt. The address of the procedure is taken from the memory whose address is four times the type number.

INTO Instruction :

If the overflow flag is set, this instruction will cause the 8086 to do an indirect far call to a procedure you write to handle overflow condition. To do call the 8086 will read a new value for IP from address 00010H and a new value of CS from address 00012H.

IRET Instruction :

The IRET instruction is used at the end of the interrupt service routine to return execution to the interrupted program. The 8086 copies return address from stack into IP and CS registers and the stored value of flags back to the flag register.

Note : The RET instruction does not copy the flags from the stack back to the flag register.

3.12 Sign Extension Instructions

CBW : Convert Signed Byte to Signed Word

This instruction copies the sign of a byte in AL to all the bits in AH. CBW does not affect any flag.

Example :

```
        ; AX = 0000 0000 1001 1010
CBW     ; convert signed byte in AL to signed word in AX
        ; Result : AX = 1111 1111 1001 1010
```

CWD : Convert Signed Word to Signed Double Word.

This instruction copies the sign bit of a word in AX to all the bits of the DX register CWD does not affect any flag.

Example :

```
        ; DX = 0000 0000 0000 0000
        ; AX = 1001 0000 1001 0001
CWD     ; Convert signed word in AX to signed
        ; doubleword in DX : AX
        ; Result :  DX = 1111 1111 1111 1111
                    AX = 1001 0000 1001 0001
```

3.13 Assembler Directives

There are some instructions in the assembly language program which are not a part of processor instruction set. These instructions are instructions to the assembler, linker, and loader. These are referred to as **pseudo-operations** or as **assembler directives.** The assembler directives enable us to control the way in which a program assembles and lists. They act during the assembly of a program and do not generate any executable machine code.

There are many specialized assembler directives. Let us see the commonly used assembler directive in 8086 assembly language programming.

ALIGN : The align directive forces the assembler to align the next segment at an address divisible by specified divisor. The general format for this directive is as shown below.

ALIGN number

where number can be 2, 4, 8 or 16.

Example :
```
ALIGN 8  ; This forces the assembler to align the next segment
         ; at an address that is divisible by 8. The assembler fills
         ; the unused bytes with 0 for data and NOP instructions
         ; for code.
```

Usually ALIGN 2 directive is used to start the data segment on a word boundary and ALIGN 4 directive is used to start the data segment on a double word boundary.

ASSUME : The 8086, at any time, can directly address four physical segments which include a code segment, a data segment, a stack segment and an extra segment. The 8086 may contain a number of logical segments. The ASSUME directive assigns a logical segment to a physical segment at any given time. That is, the ASSUME directive tells the assembler what addresses will be in the segment registers at execution time.

Example : ASSUME CS : code, DS : Data, SS : stack.

.CODE : This directive provides shortcut in definition of the code segment. General format for this directive is as shown below.

.code [name]

The name is optional. It is basically specified to distinguish different code segments when there are multiple code segments in the program.

.DATA : This directive provides shortcut in definition of the data segment.

DB, DW, DD, DQ, and DT : These directives are used to define different types of variables, or to set aside one or more storage locations of corresponding data type in memory. Their definitions are as follows :

DB - Define Byte

DW - Define Word

DD - Define Doubleword

DQ - Define Quadword

DT - Define Ten Bytes

Example :
```
AMOUNT DB 10H, 20H, 30H, 40H   ; Declare array of 4 bytes named
                               ; AMOUNT
MES DB 'WELCOME'               ; Declare array of 7 bytes and
                               ; initialize with ASCII codes for letters in
                               ; WELCOME.
```

DUP : The DUP directive can be used to initialize several locations and to assign values to these locations.

Format : Name Data_Type Num DUP (value)

Example :

```
TABLE DW 10 DUP (0)        ; Reserve an array of 10
                           ; words of memory and initialize all 10
                           ; words with 0. Array is named TABLE.
```

END : The END directive is put after the last statement of a program to tell the assembler that this is the end of the program module. The assembler ignores any statement after an END directive.

EQU : The EQU directive is used to redefine a data name or variable with another data name, variable, or immediate value. The directive should be defined in a program before it is referenced.

Formats :

Numeric Equate : name EQU expression

String Equate : name EQU <string>

Example :

```
PORT EQU 80                          ; Numeric value
NUM EQU <'Enter the first number :'>
MES DB NUM                           ; Replace with string
```

EVEN : EVEN tells the assembler to advance its location counter if necessary so that the next defined data item or label is aligned on an even storage boundary. This feature makes processing more efficient on processors that access 16 or 32 bits at a time.

Example :

```
EVEN LOOKUP DW 10 DUP (0)    ; Declares the array of ten words
                             ; starting from even address.
```

EXTRN : The EXTRN directive is used to inform assembler that the names or labels following the directive are in some other assembly module. For example, if you want to call a procedure which is in a program module assembled at a different time, you must tell the assembler that the procedure is EXTRN. The assembler will then put information in the object code file so that the linker can connect the two modules together. For a reference it is necessary to specify whether the label is near or far.

Note : Names and labels referred to as external in one module must be declared public.

Example :

```
CALLING PROGRAM          CALLED PROGRAM
DATA SEGMENT             EXTRN VAR : FAR
PUBLIC VAR               DATA SEGMENT
VAR DW                      . .
```

```
. .                                MOV AX, VAL
DATA ENDS                          . .
. .                                DATA ENDS
```

GROUP : A program may contain several segments of the same type (code, data, or stack). The purpose of the GROUP is to collect them all under one name, so that they reside within one segment, usually a data segment.

Format : Name GROUP Seg-name, . . . , Seg-name.

Example :

```
SEG GROUP SEG1, SEG2
SEG1 SEGMENT PARA 'DATA'
ASSUME DS : SEG
. . .
SEG1 ENDS
SEG2 SEGMENT PARA 'DATA'
ASSUME DS : SEG
. . .
SEG2 ENDS
```


LABEL : Assembler uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The LABEL directive is used to give a name to the current value in the location counter. The label directive can be used to specify destination for jump or call instruction or to specify reference to a data item. When label is used as destination for a jump or a call, then the label must be specified as type near or as type far. When label is used to refer a data item it must be specified as type byte, type word, or type double word.

Example :

```
NEXT    LABEL FAR      ; Can jump to NEXT from
                       ; another segment
NEXT :  MOV AX, BX     ; Cannot do far jump directly to a label
                       ; with a colon.
                       ; Initialization of stack pointer using
                       ; label directive
```

LENGTH : It is an operator which tells the assembler to determine the number of elements in some named data item such as a string or array.

Example :

```
MOV BX, LENGTH STRING1        ; Loads the Length of string in BX
```

MACRO and ENDM : The macros in the programs can be defined by MACRO directive. ENDM directive is used along with the MACRO directive. ENDM defines the end of the macro.

.MODEL : It is available in MASM version 5.0 and above. This directive provides shortcuts in defining segments. It is initializes memory model before defining any segment. The memory model can be SMALL, MEDIUM, COMPACT or LARGE. We can choose the memory model based on our requirement by referring following table.

Model	Code segments	Data segments
Small	One	One
Medium	Multiple	One
Compact	One	Multiple
Large	Multiple	Multiple

Table 3.5

NAME : The name directive is used at the start of a source program to give specific names to each assembly module.

OFFSET : It is an operator which tells the assembler to determine the offset or displacement of a named data item (variable) from the start of the segment which contains it.

Example :

```
MOV AX, OFFSET MES1     ; Loads the offset of variable, MES1 in
                        ; AX register
```

ORG : The assembler uses a location counter to account for its relative position in a data or code segment.

Format : ORG expression

Example : ORG 1000H, Set the location counter to 1000H

PTR : PTR is used to assign a specific type to a variable or to a label. It is also used to override the declared type of a variable.

Example :

```
WORD_LEN DW
  . .
  . .
  . .
MOV BL, BYTE PTR WORD_LEN   ; Byte accesses byte from word
```

PAGE : The PAGE directive helps to control the format of a listing of an assembled program. At the start of a program the PAGE directive specifies the maximum number of lines to list on a page and the maximum number of characters on a line.

Format : PAGE [length], [width]

Example : PAGE 52, 132 ; 52 lines per page and 132 characters per line.

PROC and ENDP :

PROC : The procedures in the programs can be defined by PROC directive. The procedure name must be present, must be unique, and must follow naming conventions for the language. After the PROC directive the term NEAR or FAR are issued to specify the type of the procedure.

Example : FACT PROC FAR ; Identifies the start of a procedure named FACT and tells the assembler that the procedure is far (in a segment with a different name from that which contains the instruction which calls the procedure)

ENDP : ENDP directive is used along with the PROC directive. ENDP defines the end of the procedure.

PUBLIC : Large programs are usually written as several separate modules. Each module is individually assembled, tested and debugged. When all the modules are working correctly, their object code files are linked together to form the complete program. In order for the modules to link together correctly, any variable name or label referred to in other modules must be declared public in the module where it is defined. The PUBLIC directive is used to tell the assembler that a specified name or label will be accessed from other modules.

Format : PUBLIC Symbol [. . . .]

Example : PUBLIC SETPT ; Makes SETPT available for other modules.

SEGMENT and ENDS : An assembly program in .EXE format consists of one or more segments. The start of these segments are defined by SEGMENT directive and the ENDS statement indicates the end of the segment.

Format : name SEGMENT [options] ; Begin segment
.
.
name ENDS ; End segment

Example :
```
CODE  SEGMENT
        .
        .
        .
CODE  ENDS
```

SHORT : A short is a operator. It is used to tell the assembler that only 1 byte displacement is needed to code a jump instruction. If the jump destination is after the jump instruction in the program, the assembler will automatically reserve 2 bytes for the displacement. Using the short operator saves 1 byte of memory by telling the assembler that it only needs to reserve 1 byte for this particular jump. The short operator should be used only when the destination is in the range of –128 bytes to +127 bytes from the address of the instructions after the jump.

Example : JMP SHORT NEAR_LABEL

.STACK : This directive provides shortcut in definition of the stack segment. General format for this directive is as shown below.

.stack [size]

The default size is 1024 bytes.

Example : .STACK 100 ;This reserves 100 bytes for the stack operation.

When stack is not used in the program .stack command can be omitted. This will reserve in the warning message "no stack segment" after linking the program. This warning may be ignored.

TITLE : The TITLE directive help to control the format of a listing of an assembled program. TITLE directive causes a title for a program to print on line 2 of each page of the program listing. Maximum 60 characters are allowed as title.

Format : TITLE text

Example : TITLE Program to find maximum number

TYPE : It is an operator which tells assembler to determine the type of specified variable. Assembler determines the type of specified variable in number of bytes. For byte type variable the assembler gives a value of 1. For word type variable the assembler gives a value of 2 and for double word type variable the assembler gives a value of 4.

3.13.1 Summary of Assembler Directives

Directive	Action
ALIGN	aligns next variable or instruction to byte which is multiple of operand
ASSUME	selects segment register(s) to be the default for all symbol in segment(s)
COMMENT	indicates a comment
DB	allocates and optionally initializes bytes of storage
DW	allocates and optionally initializes words of storage
DD	allocates and optionally initializes doublewords of storage
DQ	allocates and optionally initializes quadwords of storage
DT	allocates and optionally initializes 10 byte-long storage units
END	terminates assembly; optionally indicates program entry point
ENDM	terminates a macro definition
ENDP	marks end of procedure definition
ENDS	marks end of segment or structure
EQU	assigns expression to name
EVEN	aligns next variable or instruction to even byte
EXITM	terminates macro expansion

EXTRN	indicates externally defined symbols
LABEL	creates a new label with specified type and current location counter
LOCAL	declares local variables in macro definition
MACRO	starts macro definition
.MODEL	specifies mode for assembling the program.
ORG	sets location counter to argument
PAGE	sets length and width of program listing; generates page break
PROC	starts procedure definition
PTR	assigns a specific type to a variable or to a label
PUBLIC	identifies symbols to be visible outside module
TITLE	defines the program listing title

Table 3.6

3.13.2 Variables, Suffix and Operators

Variable : A variable is an identifier that is associated with the first byte of data item. In assembly language statement : COUNT DB 20H, COUNT is a variable.

Example : Array DB 10, 20, 30, 40, 50

Here, array is the variable which is associated with the first byte of the data item, i.e. 10.

Suffix : In assembly language programming base of the number of indicated by a suffix as follows :

- B - Binary
- D - Decimal
- O - Octal
- H - Hexadecimal

The default is decimal. The first digit in a hexadecimal number must be 0 through 9; therefore, if the most significant digit is a letter (A-F), then it must be prefixed with a 0.

Examples : 1010 B = 1010_2

2967 D = 2967 = 2967_{10}

3F2AH = $2F2A_{16}$

0B129H = $B129_{16}$

Operators : Arithmetic operators : "+", "–", "*", and "/".

Logical operators : "AND", "OR", "NOT, and "XOR".

Logical operators are specially used for binary operands.

3.13.3 Accessing a Procedure and Data from another Assembly Module

As mentioned earlier, usually a large program is divided into a series of modules. Each module is individually written, assembled, and tested. The object code files for the modules are then linked together to generate a linked file or executable file.

In order for a linker to be able to access data or a procedure in another assembly module correctly we have to use two assembly language directives : PUBLIC and EXTRN.

In the module where a variable or procedure is declared we must use the PUBLIC directive to let the linker know that the variable or procedure can be accessed from other modules.

In a module which calls a procedure or accesses a variable in another module, we must use the EXTRN directive to let the assembler know that the procedure or variable is not in this module but it has to access from another module. The EXTRN statement also gives the linker some needed information about the procedure. For example : EXTRN ROUTINE : FAR, TOKEN : BYTE tells the linker that ROUTINE is a FAR procedure and TOKEN is a variable of type byte.

Example 3.1 : *"File1.asm" contains a program segment which calls a subroutine (procedure) in "File2.asm". Give the necessary declarations in File1.asm and "File2 asm" (to make the subroutine of file2.asm available to file1.asm which is not locally available) and the assembling and linking to obtain the executable file.*

Solution :

File1.asm

```
EXTRN   ROUTINE : FAR
```

File2.asm

```
PUBLIC ROUTINE PROC FAR
        :
        :
ROUTINE ENDP
```

3.14 Assembly Language Programming

A program is a set of instructions arranged in the specific sequence to do the specific task. It tells the microprocessor what it has to do. The process of writing the set of instructions which tells the microprocessor what to do is called "**Programming**". In other words, we can say that programming is the process of telling the processor exactly how to solve a problem. To do this, the programmer must "**speak**" to the processor in a language which processor can understand.

Steps Involved in Programming

- **Specifying the problem :** The first step in the programming is to find out which task is to be performed. This is called specifying the problem. If the programmer does not understand what is to be done, the programming process cannot begin.
- **Designing the problem-solution :** During this process, the exact step by step process that is to be followed (program logic) is developed and written down.

- **Coding :** Once the program is specified and designed, it can be implemented. Implementation begins with the process of coding the program. Coding the program means to tell the processor the exact step by step process in its language. Each processor has a set of instructions. Programmer has to choose appropriate instructions from the instruction set to build the program.
- **Debugging :** Once the program or a part of program is coded, the next step is debugging the code. Debugging is the process of testing the code to see if it does the given task. If program is not working properly, debugging process helps in finding and correcting errors.

To write a program, programmer should know :

- How to develop program logic?
- How to tell the program to the processor?
- How to code the program?
- How to test the program?

Flow Chart

To develop the programming logic programmer has to write down various actions which are to be performed in proper sequence. The flow chart is a graphical tool that allows programmer to represent various actions which are to be performed. The graphical representation is very useful for clear understanding of the programming logic.

The Fig. 3.21 shows the graphic symbols used in the flow chart.

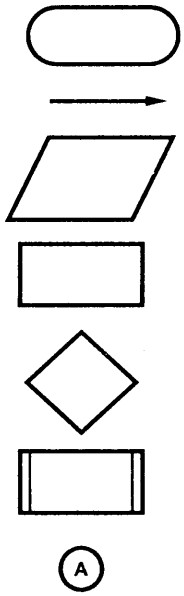


Fig. 3.21 Graphic symbols used in flow chart

Oval : It indicates start or stop operation.

Arrow : It indicates flow with direction.

Parallelogram : It indicates input/output operation.

Rectangle : It indicates process operation.

Diamond : It indicates decision making operation.

Double sided rectangle : It indicates execution of pre-defined process (subroutine).

Circle with alphabet : It indicates continuation.

A: Any alphabet

The Fig. 3.22 shows sample flow chart.

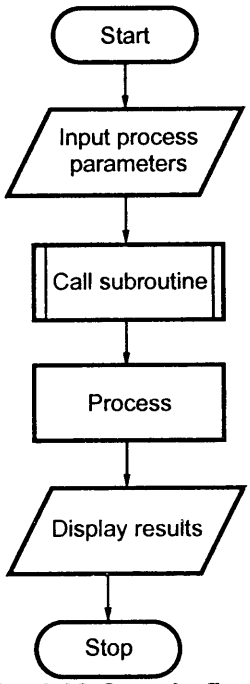


Fig. 3.22 Sample flowchart

3.14.1 Assembly Language Programs

A program which has simply a sequence of the binary codes for the instructions is called machine level language program. This binary form of the program is referred to as machine language because it is the form required by the machine. However, to write a program in machine language, programmer has to memorize the thousands of binary instruction codes for a processor. This task is difficult and error prone.

To make programming easier, usually programmers write programs in assembly language. They then translate the assembly language program to machine language so that it can be loaded into memory and executed. Assembly language uses two, three or four letter words to represent each instruction types. These words are referred to as **mnemonics**. The letters in an assembly language mnemonic are usually initials or a shortened form of the English word(s) for the operation performed by the instruction. For example, the mnemonic for addition is ADD, the mnemonic for logic AND operation is AND, and the mnemonic for the instruction for copy data from one location to another is MOV. Therefore, the meaning expressed by mnemonics help us to remember the operation performed by the instruction.

Assembly language statements are usually written in a standard form and assembly language has its own unique syntactical structure, such as requiring upper case or lower case, or requiring colons after label definitions. Here we discuss the common features that assembler shares.

The assembly text is usually divided into fields, separated by spaces and tabs. A format for a typical line from assembly language program can be given as

Label : Mnemonic Operand1, Operand2 ; Comment

The first field, which is optional, is the label field, used to specify symbolic labels. A label is an identifier that is assigned to the address of the first byte of the instruction in which it appears. As mentioned earlier, the presence of a label is optional, but if present, the label provides a symbolic name that can be used in branch instructions to branch to the instruction.

The second field is mnemonic, which is compulsory. All instructions must contain a mnemonic. The third and following fields are operands. The presence of the operands depends on the instruction. Some instructions have no operands, some have one, and some have two. If there are two operands, they are separated by a comma.

The last field is a comment field. It begins with a delimiter such as the semicolon and continues to the end of the line. The comments are for our benefits, they tell us what the program is trying to accomplish. Fig. 3.23 shows a typical 8086 assembly language instruction.

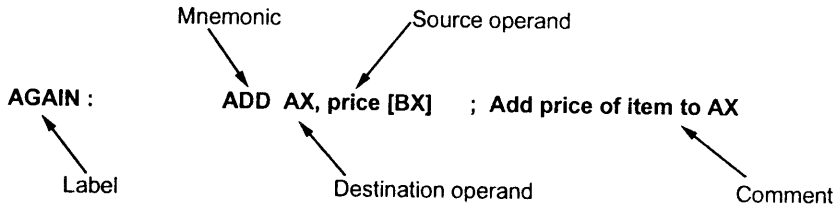


Fig. 3.23 Typical assembly language instruction

The Table 3.7 shows the comparison between machine level and assembly languages.

Sr. No.	Machine Language	Assembly Language
1.	Language consists of binary codes which specify the operation.	Language consists of mnemonics which specify the operation.
2.	Processor dependent and hence requires knowledge of internal details of processor to write a program.	Processor dependent hence requires knowledge of internal details of processor to write a program.
3.	Programs require less memory.	Programs require less memory.
4.	Programs have less execution time.	Programs have less execution time.
5.	Program development is difficult.	Program development is simpler than machine language.
6.	It is not user friendly.	It is less user friendly.

Table 3.7 Comparison between various microcomputer languages

3.14.2 Assembly Language Programming Tips

We know that a program is a set of instructions arranged in the specific sequence to perform the specific task. For writing a program for specific task, programmer may find a number of solutions (instruction sequences). A skilled programmer has to choose an optimum solution out of them for that specific task. The technique of choosing an optimum solution is an art and we can name this as an art of assembly language programming. In this section we will see some tips regarding this with the help of examples.

- **What is an optimum solution ? :** The optimum solution is the solution which takes minimum memory space for the program and minimum time for the execution of a task. When we say memory space for the program we consider space for program storage (program length), space for data storage and space used by the stack.
- **Use of proper instructions :** Many times we come across the situation where more than one set of instructions are available to perform particular function. For example, if the function is add 01 in the BX register of 8086 we have two options : ADD BX, 0001H or INC BX. In such situations we must check the space and time for both the options and then select the option which requires less space and time. Let us see the space and time required for these two instructions. The instruction ADD BX, 0001H is 4 byte instruction and requires 4 clock cycles to execute. On the other hand, INC BX is a single byte instruction and requires 2 cycles for the execution. That is instruction INC BX requires less memory space and execution time than instruction ADD BX, 0001H. Therefore, programmer must use INC BX instruction in such situation.
- **Use of advanced instructions :** We must optimally utilize the processor capabilities. For example, when it is necessary to write a program to move a block of data from the source to destination location, a programmer may initialize a pointer to indicate source location, a pointer to indicate destination location, a counter to count the number of data elements to be transferred. After transfer of one data element from source to destination location programmer may use INC, DEC and JNZ instructions to increment source and destination pointers, decrement counter and to check whether all data elements are transferred or not, respectively.

The same task can be implemented by MOVS instruction supported by 8086. Let us see the part listing of the program with both the approaches and then we compare them.

1. Part listing of program with general approach

```
        MOV SI, 1000H      ; Initialise source pointer
        MOV DI, 2000H      ; Initialise destination pointer
        MOV CX, 0020H      ; Initialise counter
BACK :  MOV AX, [SI]       ; Get data element from source
        MOV [DI], AX       ; Store it at destination
        INC SI             ; Increment source pointer
        INC DI             ; Increment destination pointer
        DEC CX             ; Decrement counter
        JNZ BACK           ; If count is not zero, repeat
```

2. Part listing of program with MOVS instruction

```
        MOV SI, 1000H      ; Initialise source pointer
        MOV DI, 2000H      ; Initialise destination pointer
        MOV CX, 0020H      ; Initialise counter
        CLD                ; Clear direction flag
   REP  MOVSB              ; Move the entire block
```

Looking at the two programs we can easily notice that the MOVSB instruction needs neither counter decrement and jump back nor pointer update instructions. All these functions are done automatically. Because MOVSB instruction copies multiple bytes from source to destination. After each byte transfer it automatically increments SI and DI pointers by 1 (since DF is 0) and decrements count in CX register and it repeats this process until CX = 0.

In the second approach, we require less number of instructions and memory space. As number of instructions are less, fetching time required for the instructions is also saved and hence we can say that the second approach requires less memory space and less time to execute the same task. So skill programmer uses second approach.

- **Use of proper addressing modes :** We know that the different ways that a processor can access data are referred to as addressing modes. If we compare the various addressing modes regarding access time required for accessing operands, we can easily make out that the register addressing takes less time to access operand than the index and indirect addressing modes. It is obvious that when operands are available in CPU registers they are immediately available for operation; however when they are in memory we have to fetch them from memory. Fetching operands takes more time. So it is advisible to store most of the operands in the CPU registers. We know that CPU registers are limited in numbers. Therefore, when they are not enough then only we should use memory space for storing the operands.

- **Prepare documentation :** Program must provide enough information so that other users can utilize the program module without having to examine its internal structure. So along with program it is advised to give the following information.

1. Description of the purpose of the program module.
2. In case of subroutine program list of passing parameters and return value.
3. Register and memory locations used.
4. Proper comments for each instruction used.

3.14.3 Programming with an Assembler

Let us see what are the steps involved in developing and executing assembly language programs. Fig. 3.24 shows these steps. The left side of the figure shows the time period, at which each step in the overall process takes place.

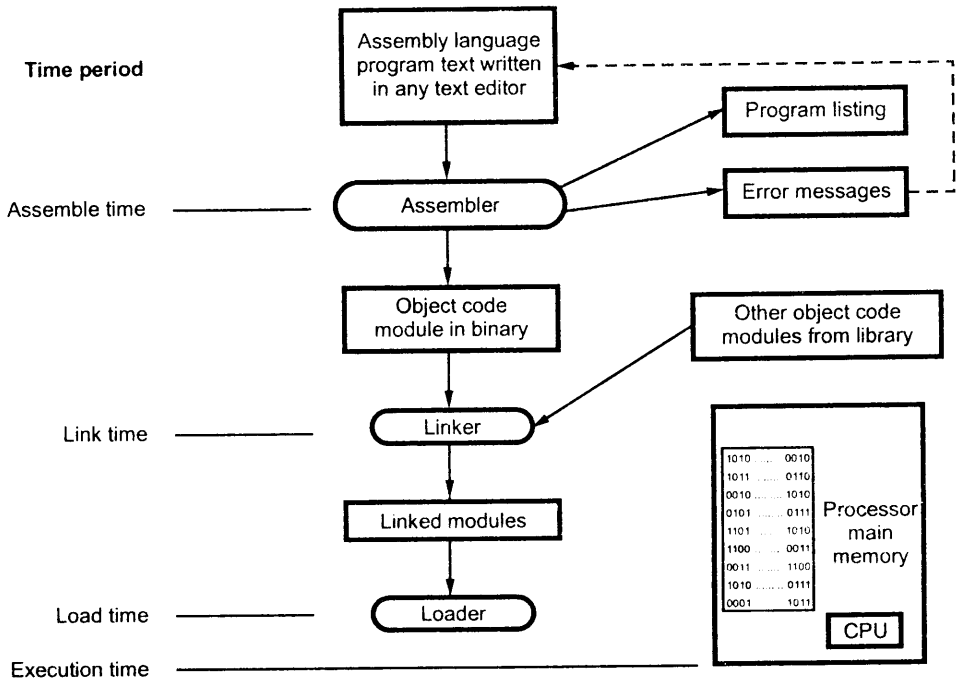


Fig. 3.24 Steps in program development and execution

The first step in the development process is to write an assembly language program. The assembly language program can be written with an ordinary text editor such as word star, edit and so on. The assembly language program text is an input to the assembler. The assembler translates assembly language statements to their binary equivalents, usually known as object code. Time required to translate assembly code to object code is called **assemble time**. During assembling process assembler checks for syntax errors and displays them before giving object code module.

The object code module contains the information about where the program or module to be loaded in memory. If the object code module is to be linked with other separately assembled modules then it contains additional linkage information. At link time, separately assembled modules are combined into one single load module, by the linker. The linker also adds any required initialization or finalization code to allow the operating system to

start the program running and to return control to the operating system after the program has completed. Most linkers allow assembly language modules to be linked with object code modules compiled from high-level languages as well. This allows the programmer to insert a time-critical assembly language routines, library modules into a program.

At load time, the program loader copies the program into the computer's main memory, and at execution time, program execution begins.

3.14.3.1 Assembling Process

As mentioned earlier, assembler translates a source file that was created using the editor into machine language such as binary or object code. The assembler reads the source file of our program from the disk where we saved it after editing. An assembler usually reads our source file more than once.

The assembler generates two files on the floppy or hard during these two passes. The first file is called the **object file**. The object file contains the binary codes for the instructions and information about the addresses of the instructions. The second file generated by the assembler is called **assembler list file**. This file contains the assembly language statements, the binary code for each instruction, and the offset for each instruction.

In the first pass, the assembler performs the following operations :

1. Reading the source program instructions.
2. Creating a symbol table in which all symbols used in the program, together with their attributes, are stored.
3. Replacing all mnemonic codes by their binary codes.
4. Detecting any syntax errors in the source program.
5. Assigning relative addresses to instructions and data.

On a second pass through the source program, the assembler extracts the symbol from the operand field and searches for it in the symbol table. If the symbol does not appear in the table, the corresponding statement is obviously erroneous. If the symbol does appear in the table, the symbol is replaced by its address or value.

We can use a suitable Editor to type .asm file. We can convert object file from .asm file using popular assemblers MASM (Microsoft macro assembler) or TASM (Turbo assembler). The command on command prompt performing this operation is as given below

C:\ MASM\ bIN\> MASM myprog.asm;

where myprog.asm is name of the .asm file which is to be converted to .obj file.

3.14.3.2 Linking Process

A linker is a program used to join together several object files into one large object file. When writing large programs, it is usually much more efficient to divide the large program into smaller modules. Each module can be individually written, tested and debugged. When all the modules work, they can be linked together to form a large functioning program.

The linker produces a link file which contains the binary codes for all the combined modules. The linker also produces a link map which contains the address information about the link files. The linker, however, does not assign absolute addresses to the program, it only assigns relative addresses starting from zero. This form of the program is said to be relocatable, because it can be put anywhere in memory to be run.

The command on command prompt for converting .obj file to .EXE file is as given below :

C : \ MASM \ BIN \ > LINK myprog.obj;

3.14.3.3 Debugging Process

A debugger is a program which allows us to load our object code program into system memory, execute the program, and debug it.

How does a debugger help in debugging a program ?

1. The debugger allows us to look at the contents of registers and memory locations after our program runs.
2. It allows us to change the contents of register and memory locations and rerun the program.
3. Some debugger allows us to stop execution after each instruction so we can check or alter memory and register contents.
4. A debugger also allows us to set a breakout at any point in our program. When we run a program, the system will execute instructions up to this breakpoint and stop. We can then examine register and memory contents to see if the results are correct at that point. If the results are correct, we can move the break point to a later point in our program. If results are not correct, we can check the program up to that point to find out why they are not correct.

In short, debugger tools can help us to isolate problems in our program.

Debug Commands

Command	Command Syntax and Description
Assembler	– A [address] A command allows you to enter the mnemonic, or human-readable, instructions directly.
Compare	– C range address C command compares two memory blocks.
Dump	– D [range] D command displays a portion of memory in hex and ASCII.
Enter	– E address [list] E command places individual bytes in memory.
Fill	– F range list F command fills a range of memory with a single value or a list of values.
GO	– G [= address] [addresses] G command execute the program in memory.
Hex	– H value 1 value 2 H command performs addition and subtraction on two hexadecimal numbers.
Load	– L [address] [drive] [first sector] [number] L command loads a file (or disk sectors) into memory.
Move	– M range address M command copies a block of data from one memory location to another.
Name	N [pathname] [arglist] N command initializes a filename (and file control block) in memory before using load or write commands.
Proceed	– P [= address] [number] P command traces the program without entering the subroutine or interrupt. If such instruction appears in the program it executes entire subroutine or interrupt routine and immediately proceeds to next instruciton in the sequence.
Quit	– Q Q command quits from debug.
Register	– R [register] R command displays the register contents on the screen.
Search	– S range list S command search a range of addresses for a list of bytes or a string.
Trace	– T [= address] [value] T command execute one or more instructions from the current CS : IP location or optional address, if specified.
Unassemble	– U [range] U command translates memory into assembly language mnemonics.
Write	– W [address] [drive] [first sector] [number] W command write a block of memory to a file or to individual disk sectors.

(See detail description of debug command in Appendix C.)

3.15 Assembly Language Example Programs

Program 1 : (Softcopy of this program, P1.asm is available at www.vtubooks.com)

```
        NAME     Addition
        PAGE     52,80
        TITLE    8086 assembly language program to add two numbers.
    .model small
    .stack 100
    .data
        No1      DB 63H          ; First number storage
        No2      DB 2EH          ; Second number storage
        Result   DW ?            ; Double byte reserved for result
    .code
    START:   MOV AX,@data        ; [ Initialises
             MOV DS,AX           ;  data segment ]
             MOV AL,NO1          ; Get first number in AL
             ADD AL,NO2          ; Add second to it
             ADC AH,00H          ; Put carry in AH
             MOV Result,AX       ; Copy result to memory
             END START
```

Program 2 : (Softcopy of this program, P2.asm is available at www.vtubooks.com)

```
        NAME     Average
        PAGE     52,80
        TITLE    8086 ALP to find average of two numbers.
    .model small
    .stack 100
    .data
        No1      DB 63H          ; First number storage
        No2      DB 2EH          ; Second number storage
        Avg          DB ?        ; Average of two numbers
    .code
    START:   MOV AX,@data        ; [ Initialises
             MOV DS,AX           ;  data segment ]
             MOV AL,NO1          ; Get first number in AL
             ADD AL,NO2          ; Add second to it
             ADC AH,00H          ; Put carry in AH
             SAR AX,1            ; Divide sum by 2
             MOV Avg,AL          ; Copy result to memory
             END START
```

Program 3 : (Softcopy of this program, P3.asm is available at www.vtubooks.com)

```
        NAME     Maximum number
        PAGE     52,80
        TITLE    8086 ALP to find maximum in the array.
    .model small
    .stack 100
    .data
```

```
   ARRAY  DB 63H,32H,45H,75H,12H,42H,09H,14H,56H,38H
                              ; Array of ten numbers
    MAX   DB 0                ; Maximum number
.code
START:   MOV AX,@data         ; [ Initialises
         MOV DS,AX            ;  data segment ]
         XOR DI,DI            ; Initialise pointer
         MOV CL,10            ; Initialise counter
         LEA BX,ARRAY         ; Initialise base pointer for array
         MOV AL,MAX           ; Get maximum number
BACK:    CMP AL,[BX+DI]       ; Compare number with maximum
         JNC SKIP
         MOV DL,[BX+DI]       ; [ If number > MAX
         MOV AL,DL            ; MAX = number ]
SKIP:    INC DI               ; Increment pointer
         DEC CL               ; Decrement counter
         JNZ BACK             ; IF count = 0 stop
                              ; otherwise go BACK
         MOV MAX,AL           ; Store maximum number
         END START
```

Program 4 : (Softcopy of this program, P4.asm is available at www.vtubooks.com)

```
     NAME Find number
     PAGE     52,80
     TITLE    8086 ALP to search a number in the array.
.model small
.stack 100
.data
  ARRAY  DB 63H,32H,45H,75H,12H,42H,09H,14H,56H,38H
                              ; Array of ten numbers
  SER_NO DB 09H               ; Number to be searched
  SER_POS    DB ?             ; Position of the searched number
.code
START:   MOV AX,@data         ; [ Initialises
         MOV DS,AX            ;  data segment ]
         MOV ES,AX
         MOV CX,000AH         ; Initialise counter
         LEA DI,ARRAY         ; Initialise base pointer for array
         MOV AL,SER_NO        ; Get the number to be searched in AL
         CLD                  ; Clear direction flag
         REPNE SCAS ARRAY     ; Repeat until match occurs or CX = 0
         MOV AL,10            ; [ Find the searched number position
         SUB AL,CL            ;  in the array if SER_POS is 0
         MOV SER_POS,AL       ;  number is not in array; otherwise
                              ;  SER_POS gives the position of
                              ;  number in the array ]
         END START
```

Program 5 : (Softcopy of this program, P5.asm is available at www.vtubooks.com)

```
     NAME     Array sum
     PAGE     52,80
```

```
    TITLE    8086 ALP to find sum of numbers in the array.
.model small
.data
     ARRAY    DB 12H,24H,26H,63H,25H,86H,2FH,33H,10H,35H
     SUM      DW 0
.code
START:   MOV AX,@data      ; [ Initialise
         MOV DS,AX         ;   data segment ]
         MOV CL,10         ; Initialise counter
         XOR DI,DI         ; Initialise pointer
         LEA BX,ARRAY      ; Initialise array base pointer
BACK:    MOV AL,[BX+DI]    ; Get the number
         MOV AH,00H        ; Make higher byte 00h
         ADD SUM,AX        ; SUM = SUM + number
         INC DI            ; Increment pointer
         DEC CL            ; Decrement counter
         JNZ BACK          ; If not 0 go to back
         END START
```

Program 6 : (Softcopy of this program, P6.asm is available at www.vtubooks.com)

```
     NAME     Separate even-odd
     PAGE     52,80
     TITLE    Separate even and odd numbers in the array.
.model small
.STACK 100
.data
     ARRAY        DB 12H,23H,26H,63H,25H,86H,2FH,33H,10H,35H
     ARR_ODD      DB 10 DUP (?)
     ARR_EVEN     DB 10 DUP (?)
.code
START:   MOV AX,@data      ; [ Initialise
         MOV DS,AX         ; data segment ]
         MOV CL,10         ; Initialise counter
         XOR DI,DI         ; Initialise odd_pointer
         XOR SI,SI         ; Initialise even_pointer
         LEA BP,ARRAY      ; Initialise array base_pointer
BACK:    MOV AL,DS:[BP]    ; Get the number
         AND AL,01H        ; Mask all bits except LSB
         JZ  NEXT          ; If LSB = 0 go to next
         LEA BX,ARR_ODD    ; [ Otherwise
         MOV AH,[BX+DI]    ; Initialise pointer to odd array
         MOV ARR_ODD,AH    ; and save number in odd array ]
         INC DI            ; Increment odd_pointer
         JMP SKIP
NEXT:    LEA BX,ARR_EVEN   ; [ Initialise pointer
         MOV AH,[BX+SI]    ;  to even array and save number
         MOV AH,ARR_EVEN   ;  in even array ]
         INC SI            ; Increment even_pointer
SKIP:    INC BP            ; Increment array base_pointer
         DEC CL            ; Decrement counter
```

```
        JNZ BACK            ; If not 0 go to back
        END START
```

It is important to note that programs discussed so far do not accept any input from keyboard and do not display any result on the video screen. This is done purposely to maintain simplicity. To accept input in various formats from keyboard and to display data on the video screen we have to use routines provided by Disk Operating System (DOS). These routines are discussed in Chapter 4. The programs given in the subsequent sections use routines provided by DOS. Therefore, students are suggested to refer these routines before further reading the remaining part of this text.

3.16 Timings and Delays

In the real time applications, such as traffic light control, digital clock, process control, serial communication, it is important to keep a track with time. For example in traffic light control application, it is necessary to give time delays between two transitions. These time delays are in few seconds and can be generated with the help of executing group of instructions number of times. This software timers are also called time delays or software delays. Let us see how to implement these time delays or software delays.

As you know microprocessor system consists of two basic components, hardware and software. The software component controls and operates the hardware to get the desired output with the help of instructions. To execute these instructions, microprocessor takes fix time as per the instruction, since it is driven by constant frequency clock. This makes it possible to introduce delay for specific time between two events. In the following section we will see different delay implementation techniques.

3.16.1 Timer Delay using NOP Instruction

NOP instruction does nothing but takes 3 clock cycles of processor time to execute. So by executing NOP instruction in between two instructions we can get delay of 3 clock cycles.

3.16.2 Timer Delay using Counters

Counting can create time delays. Since the execution times of the instructions used in a counting routine are known, the initial value of the counter, required to get specific time delay can be determined.

```
                                              Clock cycles required
        MOV CX, COUNT     ; Load count              4
BACK :  DEC CX            ; Decrement count         2
        JNZ BACK          ; If count ≠ 0, repeat    16/4
```

In this program, the instructions DEC CX and JNZ BACK execute number of times equal to count stored in the CX register. The time taken by this program for execution can be calculated with the help of clock cycles. The column to the right of the comments indicates the number of clock cycles required for the execution of each instruction. Two values are specified for the number of clock cycles for the JNZ instruction. The smaller

value is applied when the condition is not met, and the larger value is applied when it is met. The first instruction MOV CX, count is executed only once and it requires 4 clock cycles. There are count–1 passes through the loop where the condition is met and control is transferred back to the first instruction in the loop (DEC CX). The number of clock cycles that elapse while CX register is not zero are (count–1) × (2 + 16). On the last pass through the loop the condition is not met and the loop is terminated. The number of clock cycles that elapse in this pass are 2 + 4.

∴ Total clock cycles required to execute the given program

$$= \underbrace{4}_{\text{MOV CX, Count}} + \underbrace{(\text{Count}-1)\times(2+16)}_{\text{Loop}} + \underbrace{(2+4)}_{\text{Last loop}}$$

For count = 100, the number of clock cycles required are

$$= 4 + (100 - 1) \times (2 \times 16) + (2 + 4)$$

$$= 1792$$

Assuming operating frequency of 8086 system 10 MHz,

Time required for 1 clock-cycle $= \frac{1}{10\,\text{MHz}} = 0.1\ \mu\text{sec}$

∴ Total time required for execution of a given program with count equal to 100 is 179.2 μsec (1792 × 0.1).

In the above example, we have calculated the time required for the execution of program or delay introduced by the program when count value is given. However, in most of the situations we know the waiting time or delay time and it is necessary to determine what count should be loaded in the CX register to get the specified delay. Let us consider that we have to generate a delay of 50 ms using an 8086 system that runs at 10 MHz frequency. Then using same program we can calculate the count value as follows :

Step 1 : Calculate the number of required clock cycles

$$\text{Number of required clock cycles} = \frac{\text{Required delay time}}{\text{Time for 1 - clock cycle}}$$

$$= \frac{50\ \text{ms}}{0.1\ \mu\text{s}} = 500\ 000$$

Step 2 : Find the required count

$$\text{Count} = \frac{\text{Number of required clock cycles} - 4 - (2+4)}{\text{Execution Time for one loop}} + 1$$

$$= \frac{500\,000 - 4 - 6}{(16+2)} + 1$$

$$\approx 27778 = 6\text{C82H}$$

3.16.3 Timer Delay using Nested Loops

In this program one more external loop is added to execute the internal loop multiple times. So that we can get larger delays. The inner loop is nothing but the program we have seen in the last section.

```
        MOV BX, Multiplier count   ; Load multiplier count
REPE :  MOV CX, COUNT              ; Load count
BACK :  DEC CX                     ; Decrement count
        JNZ BACK                   ; If count ≠ 0, repeat
        DEC BX                     ; Decrement multiplier count
        JNZ REPE                   ; If not zero repeat
```

In the delay calculations of nested loops, the delay introduced by inner loop is very large in comparison with the delay provided by MOV BX, COUNT, DEC BX and JNZ instructions. Therefore, it is not necessary to consider the last loop for the external loop delay calculations separately. The inner loop delay calculations will remain as it is.

∴ Total clock cycles required to execute the given program

$$= [\underbrace{4}_{\text{MOV CX,Count}} + \underbrace{(\text{count}-1)\times(2+16)}_{\text{Loop}} + \underbrace{(2+4)}_{\text{Last loop}}] \times \text{multiplier count}$$

For count = 100 and multiplier count 50, the number of clock cycles required are

$$= [4 + (100 - 1) \times (2 + 16) + (2 + 4)] \times 50$$

$$= 89600$$

Assuming operating frequency of 8086 system 10 MHz,

Total time required for execution of a given program

$$= 89600 \times 0.1\ \mu\text{sec} = 8.96\ \text{ms}$$

Example 3.2 : *Write an 8086 ALP to generate a delay of 100 ms, if 8086 system frequency is 10 MHz.*

Solution :

Program :

```
        MOV CX, COUNT      ; 4
BACK :  DEC CX             ; 2
        JNZ BACK           ; 16/4
```

Step 1 : Calculate the number of required clock cycles

$$\text{Number of required clock cycles} = \frac{\text{Required delay time}}{\text{Time for 1 clock cycle}}$$

$$= \frac{100\ \text{ms}}{0.1\ \mu\text{s}} = 1000000$$

Step 2 : Find the required count

$$\text{Count} = \frac{\text{Number of required clock cycles} - 4 - (2+4)}{\text{Execution time for one loop}} + 1$$

$$= \frac{1000\,000 - 4 - 6}{(16+2)} + 1$$

$$= 55556 = \text{D904H}$$

Example 3.3 : *Write an 8086 ALP to generate a delay of 1 minute if 8086 system frequency is 10 MHz.*

Solution :

```
Program :
MOV BX, multiplier count
REPE :   MOV CX, Count          ; 4
BACK :   DEC CX                 ; 2
         JNZ BACK               ; 16/4
         DEC BX
         JNZ REPE
```

Step 1 : Calculate the delay generated by inner loop with maximum count (FFFFH)

Delay generated by inner loop for count (FFFFH = 65535)

$$= [4 + (65535 - 1) \times (2 + 16) + (2 + 4)] \times 0.1\ \mu s$$

$$= 118.1422 \text{ msec}$$

Step 2 : Calculate the multiplier count to get delay of 1 minute

$$\text{multiplier count} = \frac{\text{Required delay}}{\text{Delay provided by inner loop}}$$

$$= \frac{1 \times 60 \text{ sec}}{118.1422 \text{ m sec}}$$

$$\approx 509 = \text{1FDH}$$

3.17 Data Conversions

Before going to write and execute any assembly language program on a computer we must understand which type of data processor understands and which type of data user understands, and how they communicate with each other. User communicates with computer using input devices and computer gives outcome of process or result on the display devices or hardcopy devices such as printer or plotter. Most commonly used input device is keyboard and most commonly used output device is a display device, video monitor. These devices understand the information in ASCII format. Keyboard gives the pressed key number or character in its ASCII equivalent and for display certain number or character we have to send the ASCII equivalent of the number or character to the display

device. On the other hand, processor does not understand the ASCII format. It uses binary numbers. Therefore, it is necessary to convert input from keyboard to its binary equivalent (ASCII to binary conversion) and convert processed data by processor into ASCII format for the display (binary to ASCII conversion). Let us see how we can perform these conversions. In this section we study the routines for these conversions. Once we understand these routines we can use these routines to accept input using keyboard and to display data on video monitor.

3.17.1 Routines to Convert Binary to ASCII

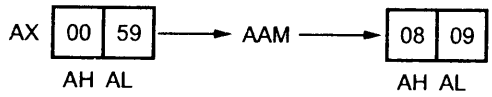
There are two ways to convert binary number into its ASCII equivalent :

- By the AAM instruction if the number is less than 100.
- By a series of decimal divisions (divide by 10).

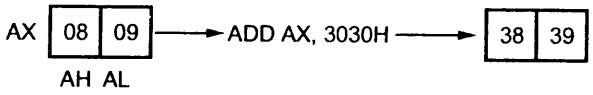
3.17.1.1 By AAM Instruction (For Number Less than 100)

The AAM instruction converts the value in AX into a two-digit unpacked BCD number in AX. For example, if number in AX is 0059H (89 decimal) before execution of AAM instruction, AX contains 0809H after execution of AAM instruction. Now we can get ASCII equivalent by adding 3030H to AX.

Algorithm :



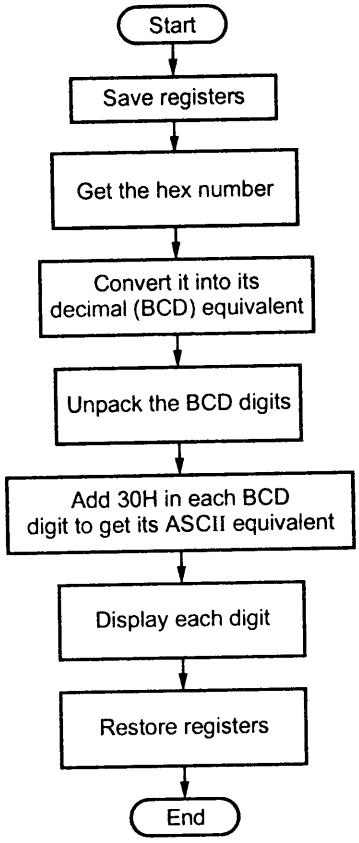
Note : 59H ⟶ 89 Decimal



Note : 38H and 39H are the ASCII equivalents of 8 and 9 respectively

1. Save contents of all registers which are used in the routine.
2. Get the data in AL register and make AH equal to 00.
3. Use AAM instruction to convert number in its decimal equivalent in the unpacked format.
4. Add 30H in each digit to get its ASCII equivalent.
5. Display digit one by one using function 2 of INT 21H.
6. Restore contents of registers.

Flowchart



Routine : Convert Binary to ASCII for number less than 100

Passing Parameter : Hex number in AL register.

```
; Routine to convert binary number into its
; decimal and then ASCII equivalent, and then display the number

BTA PROC NEAR

     PUSH DX            ; Save registers
     PUSH BX
     PUSH AX

     MOV AH, 00H        ; Clear AH
     AAM                ; Convert to BCD
     ADD AX, 3030H      ; Convert to ASCII
     MOV BX,AX          ; Save result
     MOV DL, BH         ; Load first digit (MSD)
     MOV AH, 02         ; Load function number
     INT 21H            ; Display first digit (MSD)
     MOV DL, BL         ; Load second digit (LSD)
```

```
    INT 21H                 ; Display second digit (LSD)

    POP AX                  ; Restore registers
    POP BX
    POP DX
    RET
    ENDP
```

Sample Program

```
; Sample program to convert binary number into its
; decimal and then ASCII equivalent, and then display the number

.MODEL SMALL                ; Select SMALL mode
.STACK 100                  ; Initialization of stack
.CODE

    MOV AL,59H              ; Load number in AL
    CALL BTA                ; Call routine
    MOV AH, 4CH             ; [Exit
    INT 21H                 ;  to DOS]
```

```
BTA PROC NEAR

    PUSH DX                 ; Save registers
    PUSH BX
    PUSH AX

    MOV AH, 00H             ; Clear AH
    AAM                     ; Convert to BCD
    ADD AX, 3030H           ; Convert to ASCII
    MOV BX,AX               ; Save result
    MOV DL, BH              ; Load first digit (MSD)
    MOV AH, 02              ; Load function number
    INT 21H                 ; Display first digit (MSD)
    MOV DL, BL              ; Load second digit (LSD)
    INT 21H                 ; Display second digit (LSD)

    POP AX                  ; Restore registers
    POP BX
    POP DX
    RET
    ENDP
    END
```

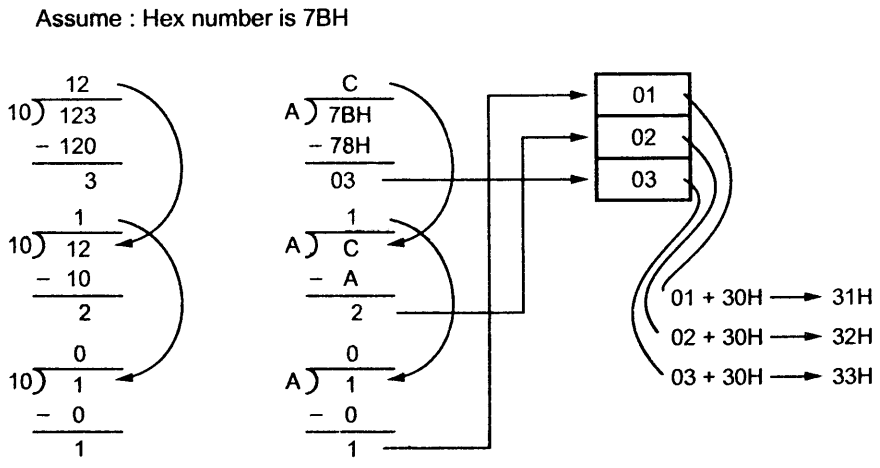
```
C:\tasm\tasm s_bta.asm
Turbo Assembler  Version 3.0  Copyright (c) 1988, 1991 Borland
International
Assembling file:      s_bta.asm
Error messages:       None
```

```
Warning messages:      None
Passes:                   1
Remaining memory:      410k

C:\tasm\tlink s_bta.obj
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International
C:\tasm\s_bta
89
```

3.17.1.2 By Series of Decimal Division

If number is greater than 99 we cannot use AAM instruction to convert given number in the BCD format. In such case we use scheme of dividing by 10 to convert any whole number from binary to an ASCII coded character string that can be displayed on the video monitor.

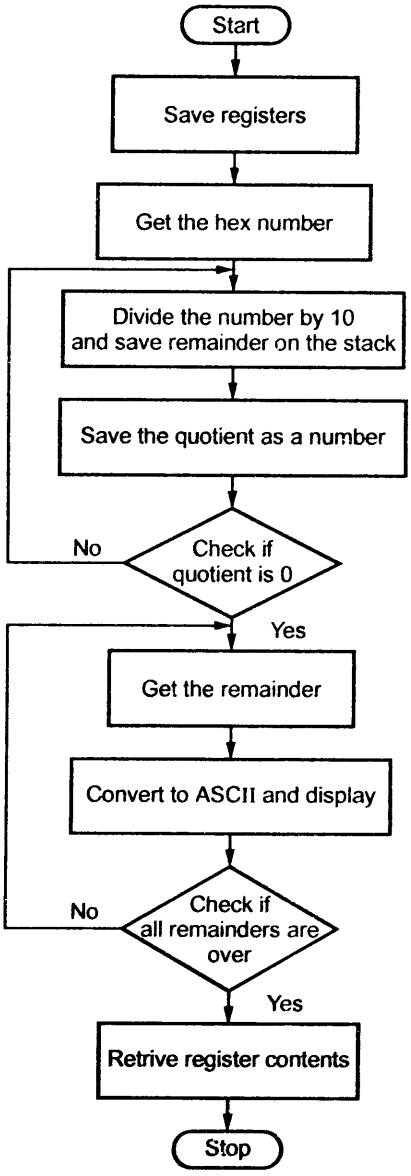


Let us see the algorithm for converting number from binary to ASCII code.

Algorithm

1. Save contents of all registers which are used in the routine.
2. Divide the number by 10 and save the remainder on the stack as a significant BCD digit.
3. Save the quotient as a number.
4. Repeat step 1 and 2 until quotient is 0.
5. Retrieve each remainder from stack and add 30 H to convert to ASCII before displaying or printing.
6. Restore contents of registers.

Flowchart



Routine : Convert Binary to ASCII

Passing parameter : 4-digit hex number in AX register.

```
; Routine to convert 4-digit hex into its decimal
; and then to ASCII equivalent, and display it

BTA4D PROC NEAR

        PUSH DX              ; Save registers
        PUSH CX
```

```
        PUSH BX
        PUSH AX
        MOV CX, 0           ; Clear digit counter
        MOV BX, 10          ; Load 10 decimal in BX
BACK:   MOV DX, 0           ; Clear DX
        DIV BX              ; Divide DX : AX by 10
        PUSH DX             ; Save remainder
        INC CX              ; Counter remainder
        OR AX, AX           ; Test if quotient equal to zero
        JNZ BACK            ; If not zero divide again
        MOV AH, 02H         ; Load function number
DISP:   POP DX              ; Get remainder
        ADD DL, 30H         ; Convert to ASCII
        INT 21H             ; Display digit
        LOOP DISP
        POP AX              ; Restore registers
        POP BX
        POP CX
        POP DX
        RET
        ENDP
        END
```

Sample Program

```
; Sample program to convert 4-digit hex into its decimal
; and then to ASCII equivalent, and display it

.MODEL SMALL            ; Select SMALL model
.STACK 100              ; Initialise stack segment
.CODE
        MOV AX,2ABCH
        CALL BTA4D      ; Call routine
        MOV AH, 4CH     ; [Exit
        INT 21H         ;  to DOS]
BTA4D   PROC NEAR
        PUSH DX         ; Save registers
        PUSH CX
        PUSH BX
        PUSH AX
        MOV CX, 0       ; Clear digit counter
        MOV BX, 10      ; Load 10 decimal in BX
BACK:   MOV DX, 0       ; Clear DX
        DIV BX          ; divide DX : AX by 10
        PUSH DX         ; Save remainder
        INC CX          ; Counter remainder
        OR AX, AX       ; Test if quotient equal to zero
        JNZ BACK        ; If not zero divide again
        MOV AH, 02H     ; Load function number
DISP:   POP DX          ; Get remainder
        ADD DL, 30H     ; Convert to ASCII
```