
UNIT 13 MICROPROCESSOR ARCHITECTURE

Structure	Page No.
13.0 Introduction	
13.1 Objectives	
13.2 Structure of 8086 CPU	
13.2.1 Bus Interface Unit	
13.2.2 Execution Unit	
13.2.3 Register Set	
13.3 Instruction Set of 8086	
13.3.1 Data Transfer Instructions	
13.3.2 Arithmetic Instructions	
13.3.3 Bit Manipulation Instructions	
13.3.4 Program Execution Transfer Instructions	
13.3.5 String Instructions	
13.3.6 Processor Control Instructions	
13.4 Addressing Modes	
13.4.1 Register Addressing Mode	
13.4.2 Immediate Addressing Mode	
13.4.3 Direct Addressing Mode	
13.4.4 Indirect Addressing Mode	
13.5 Summary	
13.6 Solutions/Answers	

13.0 INTRODUCTION

In the previous three blocks of this course, you have gone through the concept of data representation, logic circuits, memory and I/O organisation, instruction set architecture, micro-operations, control unit etc. of a computer system. The processor of a general purpose computer consists of an instruction set, which uses a set of addressing modes. The control unit of a processor uses a set of registers and arithmetic logic unit to process these instructions. This unit present details of a micro-processor, in the content of all the above concepts. We have selected a simple micro-processors 8086, for the discussion. Although the processor technology is old, all the concepts are valid for higher end Intel processor family, which are commonly referred to as x86 family. This block does not attempt to make you an expert assembly programmer, however, you will be able to write reasonably good assembly programs . This unit discusses the 8086 microprocessor in some detail. This unit will introduce you to block diagram of components of 8086 microprocessor. This is followed by discussion on the register organization for this processor. Some useful instructions and addressing modes of this processor are also discussed in this unit. Please note the concepts discussed in this unit may be useful in writing good Assembly Programs.

13.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the role of various components of 8086 microprocessor;
- illustrate the use of segmentation in 8086 microprocessor;
- use some of the important instruction of 8086 microprocessor
- illustrate the use of different types of addressing modes of 8086 microprocessor.

13.2 STRUCTURE OF 8086 MICROPROCESSOR

A microprocessor contains one or more processing unit on a single chip. Today's processors contain multiple processing cores in a single chip, therefore are called multi-core processors. A computer system consists of a micro-processor, memory unit and input/output interfaces, internal and external connection structure, such as buses; and several input/output devices. The bus size of a processor is a very important design parameter. For example, the address bus of a processor, generally, can determine the size of the physical main memory. The data bus determines the size of the data that can be transferred from the memory to the processor registers.

The size of address bus of 8086 micro-processor is 20 bits and data bus is 16 bits. Thus, 8086 micro-processor has $2^{20} = 1\text{M}$ Byte base memory. From this memory, about 640 KB was part of base RAM and remaining was used as ROM.

8086 micro-processor was designed as a complex instruction set computer with the basic objectives of supporting more instructions, addressing modes and more throughput. Present day multi-core processors are far more powerful than 8086 micro-processor, but objective of this block is to introduce some of the basic features of micro-processor and assembly language programming. For the basic discussion this processor is good example.

A microprocessor executes a sequence of machine instruction, which can be represented as the following notional program:

```
repeat execution of <instruction cycle>
{
    fetch(instruction);
    execute(instruction)
        decode instruction
        fetch operand;
        execute the desired operation on data
    if (interrupt) process it and return to program execution;
}
```

The 8086 microprocessor consists of two independent units (refer to Figure 13.1):

1. The Bus Interface unit, and
2. The Execution unit.

These units can function independently, therefore, they can function as two stages instruction pipeline. Components of these two units as shown in Figure 13.1 are explained in the following sections.

13.3.1 The Bus Interface Unit

The Bus Interface Unit (BIU) is responsible for external communication through the system bus. It has a dedicated address adder circuit, which takes input from segment registers and special registers of the processor. This unit also has an instruction stream queue of 6 byte length and is used to store the instruction, which is to be executed. The main tasks of this unit are:

- Computing the physical address of the instruction or data or input/output port from/ to the information is to be read or written into.
- This unit then reads or writes the information from the physical address as computed above.
 - If an instruction is fetched, it is stored in the instruction stream queue.

- Data is fetched into a general purpose register.
- In case of writing the data value of a selected register is written into a desired memory location or I/O port .

An instruction queue is useful only if more than one instructions are fetched simultaneously, which may be used for instruction pipelining involving the stages of instruction fetch and instruction execution.

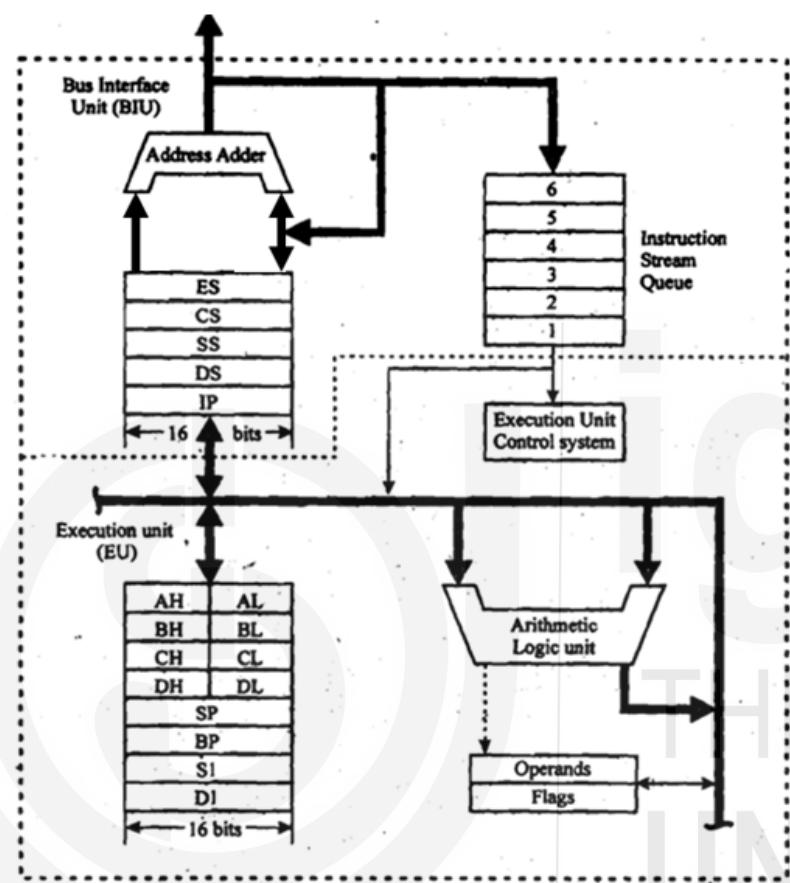


Figure 13.1: Structure of Intel 8086 Microprocessor

The Segment Registers

8086 microprocessor uses a very interesting concept of segmentation. As stated earlier that a 8086 microprocessor has 20 bit address but and 16 bit data bus. Now, assume that direct addressing scheme is used to address an operand in the memory. Since, the size of data is 16 bits, therefore, an efficient system will have a direct memory address of 16 bits, as this is the amount of data that can be fetched in one memory read operation. However, the address of an operand has to be 20 bits long (size of address bus). Thus, the two design options are either fetch two data words, which will be every inefficient; or use a concept of segmentation, which was specifically designed to this microprocessor.

The BIU of this microprocessor has four specific segment registers, namely CS: Code Segment register, DS: Data Segment register, SS: Stack Segment register, and ES: Extra Segment register. All of these registers are 16 bit long. Why segmentation? Segmentation divides the 1 MB memory of the computer associated with this microprocessor into logical overlapping segments of 64 KB. A program can have several code, data and stack segments. However, a maximum of four segments, one each of each type, may be available for accessing data and instructions at a specific

time, as there are four segment registers only. Thus, a program can consist of logical segments of code, data, stack etc. Thus, address of a data byte stored in the memory consists of a double *Segment Register (16 bits)*: *Offset Register (16 bits)* Pair. How does this segmented addressing better than fetching two words? In the segmented scheme an address included in an instruction consists of only the 16 bit memory address, thus, a segment can be a maximum size of $2^{16} = 64$ KB only. In addition, as the size of segment register is 16 bits, therefore, there can be $2^{16} = 65536$ number of segments. Please note that these segments will be overlapping as the size of base memory for this processor is only 1 MB. Figure 13.2 shows the memory organisation of 8086 microprocessor. Thus, a segment register is loaded with the address of current segments and offset is used to represent data within that segment. Thus, instruction just needs to store 16 bit address only. The address adder computes the 20 bit physical address from the *Segment Register (16 bits)*: *Offset Register (16 bits)* Pair. Please note all the content in Figure 13.2 is in hexadecimal notation.

Segment Start Address	Offset Address	Physical Memory Address	Byte content of Memory
0000 Offsets in this segment: 0000 to 001F Segment size (in byte) = 0020 in hexadecimal Which is 32 in decimal	0000	00000	52
	0001	00001	AA
	0002	00002	24

	000E	0000E	CF
	000F	0000F	32
	0010	00010	32
	0011	00011	66

	001E	0001E	--
	001F	0001F	--
	0000	00020	49
	0001	00021	23
	0002	00022	1F
0002 Offsets in this segment: 0000 to 00DF Segment size (in byte) = 00E0 in hexadecimal Which is 224 in decimal 0000 to 00DF Segment size = 00E0 in hexadecimal Which is 224 in decimal
	000E	0002E	11
	000F	0002F	42
	0010	00030	34
	0011	00031	CD

	00FE	0011E	AB
	00FF	0011F	AB

	FFFFE	1001E	DD
	FFFFF	1001F	EE

	Unused memory	FFFFC	--
		FFFFD	--
		FFFE	--
		FFFFF	--

Figure 13.2: Memory Organisation of Intel 8086 microprocessor

Figure 13.2 shows two hypothetical segments (just for illustration) in the 1MB memory using hexadecimal notation. Assume that one segment data is of 30 bytes, thus it can be accommodated in the segment of size 32 Bytes. Please note that segment start address for this segment is 0000h and offset of these locations are 0000 to 001Fh. Therefore, the second segment can start from the physical memory address 00020h. The second segment is assumed to be of 64 KB and starting from physical memory 00020h. Therefore, it has segment starting address as 0002h and offset values

from 0000h to FFFFh. An interesting fact about the memory of 8086 processor is that, although a single byte has an address, but in a single memory access two bytes are transferred through data bus. For example, access to an even memory offset 0000h will transfer bytes at offset 0000h and 0001h. However, in case, you try to access an odd memory offset like 0013h, then the bytes 0012h and 0013h would be transferred to the processor.

Now, the question is given the segment starting address of 16 bits and segment offset of 16 bits, how will you compute the physical address? The designers of 8086 used an address adder to compute physical address. The addition is performed as follows:

Given: Segment Address 0002h; and offset say 0001h

Physical address is computed by shifting segment address to left by one hexadecimal digit (appending 0 as the lowest hexadecimal digit and add the offset in the result).

The Segment Address (hexadecimal)	0	0	0	2
Shift left and add zero in least significant digit	0	0	0	2 0
Add the offset		0	0	1
Resulting 20 bit physical address	0	0	0	2 1

Given: Segment Address 0002h; and offset say FFFFh. The physical address will be computed as:

The Segment Address (hexadecimal)	0	0	0	2
Shift left and add zero in least significant digit	0	0	0	2 0
Add the offset		F	F	F
Resulting 20 bit physical address	1	0	0	1 F

Please note that F+2 will be $15+2=17$, so addend is 1 and carry is 1. Also when you add carry 1 to F, it will be 16, which is addend is 0 and carry is 1

What are the advantages of segmentation?

The following are the main advantages of segmentation.

- Since segment can overlap, therefore, in case a segment is smaller than 64 KB, next segment can start almost immediately from where the segment ends, thus, memory wastage is minimized.
- A program stores only the offset within a segment, thus, program code can be relocated to another segment, if the need so be. Thus, segmentation supports writing of reloadable programs.
- Instruction uses only 16 bit address field instead of 20 bits of address bus, thus, reducing the overall size of instructions and program.

Use of Segment Registers

As discussed earlier, 8086 microprocessor has four segment registers, they are used with specialized registers that store to compute physical address. These pairs are:

- (a) Code Segment (CS) register and Instruction Pointer (IP) register, which is the offset of the next instruction to be executed, to compute the address of the next instruction to be fetched. The following example explains their use.

An assumed Code Segment(CS) Address (hexadecimal)	0	1	A	D
Shift left and add zero in least significant digit	0	1	A	D 0
Assume that IP contains an offset 1A3Ch		1	A	3 C
Resulting 20 bit physical address	0	3	5	0 C

Please note $D+3 = 13+3=16$, therefore, carry = 1 and sum hex digit = 0
 $A+A+carry(1) = 21$, so carry = 1 and sum = $21-16=5$

- (b) Stack Segment (SS) register and Stack Pointer (SP) register, which points to the top of the stack in the stack segment, to compute the address of the top of the stack. The following example explains their use.

An assumed Stack Segment(SS) Address (hexadecimal)	F	1	1	D
Shift left and add zero in least significant digit	F	1	1	D 0
Assume that SP contains an offset 0110h		0	1	1 0

Resulting 20 bit physical address	F	1	2	E	0
--	---	---	---	---	---

- (c) Data Segment (DS) register and Offset to compute the address of the data to be fetched. The following example explains their use.

An assumed Data Segment(DS) Address (hexadecimal)	A	5	8	3
Shift left and add zero in least significant digit	A	5	8	3
Assume that data offset is A021h		A	0	2
Resulting 20 bit physical address	A	F	8	5

- (d) Extra Segment (ES) register and offset to compute the address of extra data segment (in case two data segments are used at the same time).

13.3.2 Execution Unit

The execution unit of the 8086 micro-processor consists of a set of general purpose and special purpose registers, an internal data and control bus, arithmetic and logic unit (ALU) and flags registers. The size of ALU of the 8086 processor is 16 bits. The execution unit control system decodes and performs the required operation on the data. It has the following main components:

Control Circuitry for Instruction Decode and operand specification and ALU

The 8086 processor uses a micro-programmed control unit, which decodes the instruction and executes it as per the micro-program stored in the control memory. The control unit is also responsible for generating the control timing sequences. ALU performs the operation on the data as instructed by the control unit.

Registers

8086 has several kinds of registers, which includes general purpose, special purpose registers and a special flag register. The next section explains the role of different registers of 8086 micro-processor.

13.3.3 Register Set

The 8086 registers have five different categories of registers. The following table explains the role of these registers.

Register Category	Register Name and Size	Special Purpose, if any
Segment Registers	CS (16 bits)	For storing the base address of code segment
	DS (16 bits)	For storing the base address of data segment
	SS (16 bits)	For storing the base address of stack segment
	ES (16 bits)	For storing the base address of extra data segment
General Purpose Register: Can be used for any computation, in addition they are used for specific purpose as stated in this table	AX - 16 bits ; it consists of two byte register AH, which stores the higher byte and AL, which stores the lower byte	It is also called accumulator register. It can store the results of addition or subtraction operation; for some instructions like multiplication and division it store one of the operand.
	BX - 16 bits ; it consists of BH and BL	It is also called base register. It stores the base location of a memory array.
	CX - 16 bits ; it consists of CH and CL	It is also called counter register. It can be used for keeping count in looping instructions

	DX - 16 bits ; it consists of DH and DL	This register can be used for I/O operation.
Pointer and Index Registers: These registers can also be used as general purpose registers	BP (16 bits)	Base Pointer register used in stack segment
	SI (16 bits)	Source Index register used in data segment
	DI (16 bits)	Destination Index register used in extra data segment
Special Register	SP	Stack Pointer register, points to the top of the stack.
Flags Register	It consists of 16 flags set by the last ALU operations. Each flag is 1 bit long	Some of the important flags are carry flag (CF), Parity Flag (PF), Auxiliary Flag (AF), Zero flag (ZF), Sign Flag (SF), Overflow Flag (OF), Interrupt Enable flag (IF) and other control flags.

Check Your Progress 1

1. What are the different components of Bus Interface unit and what are their uses?

.....
.....
.....

2. Compute the physical address for a 8086 microprocessor for the following:

- (a) CS:IP = 0111h:0020h
 (b) DS:BX = 0211h:0100h
 (c) SS:SP = 42AAh:0123h

.....
.....
.....

3. What is the role of a flag register in 8086 microprocessor? Can it be used as general purpose register?

.....
.....
.....

13.4 INSTRUCTION SET OF 8086

In the previous sections, the basic structure of 8086 micro-processor was discussed. This section presents the instruction set of this micro-processor. This section presents only few instructions of each type, as idea is to present example of basic instructions, which you may require to write assembly programs in the next two units.

Interestingly, the 8086 instructions can be 1 byte to 6 byte long. A general format of instructions presented here is as under:

Label:	Operation mnemonic	Operand(s)	; Comment
LOOP:	ADD	AX, DX	; AX ← AX + DX

Label is optional and is used to identify an instruction. It may be used if a subgroup of instructions are to be repeated. Operation mnemonic identifies the operation to be performed. These instructions, depending on the operation, may have zero, one or two

operands. It may be noted that in 8086 micro-processor, if an instruction has two operands, then at least one of the operand has to be a register operand; or in other words both the operands cannot be memory operands. This restriction is due to limits on the size of instruction. In addition, an operand address may use several addressing modes, which are discussed in section 1.5. The comments in the instruction are optional and are written after a semi colon (;) symbol. The example of an addition instruction is shown, where LOOP: is the label, ADD operation is to be performed on operand AX an DX. Please note that the comment in this case states the nature of addition instruction in 8086 microprocessor. The 16 bit contents of AX and DX are added and by the ALU and result is stored in AX register. In addition, this operation will also set the flags register.

The following are some of the important functional groups of the 8086 instructions.

13.4.1 Data Transfer Instructions

Data transfer instructions are required for moving the data between a pair of source and destination. Following are some of the more useful 8086 data transfer instructions.

MOV instruction: MOV destination, source (transfers source data to destination)

This instruction transfers the data from source to destination. The source or destination can be a general purpose register, immediate operand, memory location or I/O port. However, it may be noted that both the source and destination cannot be memory location in one instruction.

Example:

To move an immediate operand 2F1Ch into DX register, you can use the following instruction:

MOV DX, 2F1Ch

To move content of DX register into AX register, you can use the following instruction:

MOV AX, DX

PUSH and POP instructions: PUSH operand or POP destination

PUSH and POP instructions are used to transfer a word (2 bytes) to and from the sword stack of 8086 microprocessor. The stack in 8086 microprocessor grows from a higher memory address to lower memory address as shown in Figure 13.3

	Offset	Stack content		Offset	Stack content		Offset	Stack content
	0000			0000			0000	
	0001			0001			0001	
	
	00FC		SP	00FC	FF		00FC	
	00FD			00FD	AA		00FD	
SP	00FE	AB		00FE	AB	SP	00FE	AB
-	00FF	BD		-	BD	-	00FF	BD
Initial Stack State			Let AX = AAFFh After PUSH AX			After POP DX DX = AAFFh		

Figure 13.3: Stack after one PUSH and one POP instructions.

Please note the following about Figure 13.3

- The stack is a word stack and the operands of PUSH and POP instructions are word operands. These two instructions does not affect the flag registers.

- The maximum size of this stack segment is 0100h having offsets 0000h to 00FFh. The stack segment register value is not shown.
- In 8086 microprocessor, the stack grows from higher offset to lower offset. The stack would be empty if SP contains 0100h. Stack is full when SP is 0000h.
- The PUSH instruction causes the decrementing the stack pointer by a value 2 (as stack is a word stack and the offset is an address of a byte), i.e. SP=SP-2, and then the word operand of the PUSH instruction is put in the stack locations pointed to by the SP.
- POP instruction results in moving the content at the stack location into the destination register, specified by the instruction. This is followed by incrementing the stack pointer register value by 2, i.e. SP=SP+2.

PUSHF and POPF instructions: The PUSHF instruction is used for pushing the current flags register on to the stack, while POPF pops the content at the top of the stack to fags register.

Other data transfer instructions

There are a number of other data transfer instructions. These instruction and their purpose is given in the following table:

MNEMONIC	DESCRIPTION
XCHG destination, source	Exchanges bytes of words of source and destination. At least one operand should be a register operand.
XLAT	This is a complex instruction, which translates a byte of AL register using a lookup table. This instruction uses AL register as the operand. An example of this instruction is given in Unit 15.
LEA register, source	This instruction results in loading of 16 bit effective address of source operand to the specified register operand. This instruction is useful for array index manipulation.
IN accumulator, port address	This instruction is used to transfers a byte or word from a specified Input port to accumulator register. The instruction can use DX register as implied operand for port address. The port address can also be an immediate operand.
OUT port address, Accumulator	This instruction can be used to transfer a byte or word, which is in accumulator register to specified output port address of an output devices, such as monitor or printer
LDS/LES	These instructions and used to loading data segment/extras data segment respectively along with one specified registers. Details on these instructions are beyond the scope of this unit.
LAHF/SAHF	The LAHF loads the low byte of flags register to AH register, while SAHF stores value of AH register to low byte of flags register.

13.4.2 Arithmetic Instructions

8086 microprocessor has a large number of arithmetic instructions. These instructions are explained below:

ADD and ADC instructions: ADD destination, source and ADC destination, source

The purpose of ADD instruction is to simply add the two operands and the result of addition is stored in *destination*. The source operand can be a general purpose register, immediate operand, memory location etc. the destination may be register operand or memory location. Also both the operands should not be memory operand in an

instruction. It may be noted that both source and destination operand should either be byte operands or word operands. This instruction causes changes in several flags of the flags register. Some of these flags are: carry flag, overflow flag, sign flag, zero flag etc. ADC instruction in addition to adding the source and destination also adds the Carry flag of the flags register.

.Example:

To add an immediate operand 2F1Ch into DX register, you can use the following instruction:

ADD DX, 2F1Ch

To add content of BL register into AL register, you can use the following instruction:

MOV AL, BL ; the result of addition will be in AL register

INC and DEC instructions: INC destination and DEC destination

The purpose of INC instruction to increment the destination operand by 1, while DEC instruction decrements the destination operand by 1. The operand can a memory or register operand of byte or word type. the two operands and the result of addition is stored in *destination*. This instruction causes changes in several flags of the flags register. Some of these flags are: overflow flag, sign flag, zero flag etc. ADC instruction in addition to adding the source and destination also adds the Carry flag of the flags register.

Example:

To increment the AL register content by a value 1, you may use the instruction:

INC AL

To decrement the value of BL register by 1, you may use the instruction:

DEC BL

AAA and DAA instructions:

The AAA instruction performs ASCII adjust after addition, whereas DAA instruction performs decimal, i.e. binary coded decimal, adjust after addition. These two instructions does not have any operand expect the implied operand, which is AL register. 8086 microprocessor allows you to add two decimal digits (0 to 9) stored in ASCII format, unpacked BCD format (which consists of single BCD digit in a byte) or packed BCD format (which consists of two BCD digits in a byte). AAA is used to adjust the results, if you have added ASCII digits or unpacked BCD. DAA is used to adjust the results, if you have added two packed BCD numbers. The following examples explains these two instructions.

Example:

Consider the AL register has ASCII digit '7' and BL contains ASCII '6'. You want to add these two values to get an answer 13 in decimal. One of the way would be to convert these operand into binary and perform the addition and convert the results back to desired format. Other way will be to use AAA as follows:

```
ADD AL, BL      ; Given AL = 001101112 and BL = 001101102
                  ; Result would be 011011012 (incorrect sum)
AAA             ; Result would be adjusted by adding 0110 in lower
                  ; 4 bits and setting the AF and CF flags as AL is greater than
                  ; 9. The upper four bits will be made 000000112. The CF
                  ; indicates that the result is 13.
```

Example for DAA

```
ADD AL, BL      ; Given AL = 000101112 (packed BCD 17)
                  ; and   BL = 010101102 (packed BCD 56)
DAA             ; Result      011011012 (sum 6D is incorrect)
                  ; As lower 4 bits > 9, so adjust lower 4 bits by adding 0110
```

MUL, DIV and IDIV instructions: MUL source, DIV source and IDIV source

MUL and DIV instructions are unsigned multiplication and unsigned division instructions respectively. IDIV is a signed division instruction. The source can be a memory or register operand, which contains either byte data or word data. For these instructions one of the operand is assumed to be AL register (if data is of byte type) or AX register (if data is of word type). The result of MUL instruction is stored in AX register (if data is of one byte) or DX and AX pair (if data is of word type). Thus, symbolically the MUL instructions can be represented as:

$$\begin{aligned} \text{AX} &\leftarrow \text{AL} \times \text{source} \text{ (if source is 8 bit data)} \\ \text{DX, AX} &\leftarrow \text{AX, sources} \text{ (if source is 16 bit data)} \end{aligned}$$

In case in this instruction, if most significant bit of the result is 0, then carry and overflow flags are set to 0. In case a byte is to be multiplied with a word operand, then you must first convert the byte operand to a word operand using instructions like CBW given later in the unit.

The result of DIV and IDIV instructions for byte operands is stored as AH stores remainder and AL stores quotient of division, or for word operands DX stores the remainder and AX stores the quotient. in AH register (if data is of one byte) or DX and AX pair (if data is of word type). Thus, symbolically the MUL instructions can be represented as:

$$\begin{aligned} \text{AH (Remainder) AL (Quotient)} &\leftarrow \text{AL / source} \text{ (if source is 8 bit data)} \\ \text{DX (Remainder) AX (Quotient)} &\leftarrow \text{AX / source} \text{ (if source is 16 bit data)} \end{aligned}$$

In the division operation a 0 value in the source register will result in run time error.

Example:

Assume that AL register contains 11h and BL register contains 02h.
 Multiplication and division instructions will give following results:
 MUL BL ; Result 11h × 02h = 22h; The AH = 00h and AL=22h
 DIV BL ; Result 11h / 02h = Remainder in AH= 01h and
 ; Quotient in AL 08h

CMP instructions: compares destination and source operands

This is a very interesting instruction used for comparing two operands. This instruction only sets the flag by subtracting source from the destination operand (both byte or both word). Both the source and destination operands cannot be memory operands at the same time. This operation may set carry flag zero flag, sign flag etc. The following example explains how flags may be set by this operand. This instruction only changes the flags, no operand value is changed.

Example:

Instruction	Flags if AX= CX	Flags if AX > CX	Flags if AX < CX
CMP AX, CX	CF=0; ZF=1; SF=0	CF=0; ZF=0; SF=0	CF=1; ZF=0; SF=1

Other arithmetic instructions

Some of the other instructions are given below:

SUB destination, source	This instruction subtract source from destination. The carry flag in subtraction is a borrow flag.
SUB destination, source	Subtracts with previous borrow, if any.
NEG source	Creates the 2's complement of source number.

AAS, DAS	Works in a similar manner as AAA and DAA, except they operate after subtraction operation.
AAM, AAD	Works in a similar manner as AAA, except the operation is multiplication and division respectively.
CBW, CWD	These instructions convert byte to word or word to double word respectively. The value of sign bit is filled in the upper byte or word as the case may be. For CBW operand is in AL register and resulting word is in AX register; whereas for CWD the operand is in AX register and the double word is in DX, AX pair.

13.4.3 Bit Manipulation Instructions

The Bit manipulation instructions are used to manipulate the bit wise data. These instructions are very useful in performing logical operation on the data. The following are some of the bit manipulation instructions:

NOT, AND, OR, XOR instructions:

These are logical instructions of 8086 microprocessor. NOT instruction takes only one operand, while all other instruction have destination and source operands. The operands can be memory or register operands and both the operands cannot be memory operands in a single instruction.

Example:

Let AL= 00111010₂ and BL=11011100₂

NOT AL ; the result in AL would be 11000101₂

AND AL, BL ; the result in AL would be 00011000₂

OR AL, BL ; the result in AL would be 11111110₂

XOR AL, BL ; the result in AL would be 11100110₂

TEST destination, source instruction:

This instruction performs the AND operation on the two operands, but does not changes the operands value. This instruction clears the carry and overflow flags and sign flag and zero flags are set as per the operand.

SHL and SHR; SAL and SAR; ROL and ROR; RCL and RCR instructions:

All these eight instructions are shift instructions with small difference. All these instructions take two operands *destination and count*. The count specifies the count of times the bits of the destination operand are to be shifted. The alphabet L or R at the end of instruction mnemonic specifies Left shift or Right shift respectively. Count sometimes can be stored in CL register.

Following diagram and example explains these shift operations, assuming that data is of byte type and the count of shift is by one bit:

CF	AL Register Value								SHL is shift left; SAL is arithmetic Shift left.
0	1	0	0	0	0	0	1	1	Initial Value
1	0	0	0	0	0	1	1	0	After execution of SHL AL or SAL AL
←									Direction of shift

In SHL or SAL instruction 0 is put at the least significant bit (shown in green colour) and all the bits of the operand are shifted towards the left by 1 bit. The most significant bit is shifted to carry flag.

CF	AL Register Value								SHR is logical shift right.
1	1	1	0	0	0	0	1	1	Initial Value

1	0	1	1	0	0	0	1	After execution of SHR AL
	→							

All the bits of the byte are shifted towards the right. The most significant bit gets the value 0 and least significant bit is pushed to the carry flag (shown in green colour).

CF	AL Register Value								SAR is arithmetic shift right;
0	1 1 0 0 0 0 1 1								Initial Value
1	1 1 1 0 0 0 0 1								After execution of SAR AL
	→								

In the arithmetic shift right, all the bits are shifted towards the right. The most significant bit, which is a sign bit retains the same sign (please see the 1's in the left most position in the diagram above) and least significant bit is pushed to the carry flag (shown in green colour).

CF	AL Register Value								ROL is rotate left;
0	1 0 0 0 0 0 1 1								Initial Value
1	0 0 0 0 0 1 1 1								After execution of ROL AL
	←								

In the Rotate shift left, all the bits are shifted towards the left. The most significant bit is shifted to CF as well as rotated to least significant bit, as shown above (in green colour).

CF	AL Register Value								ROR is rotate right;
0	1 0 0 0 0 0 1 1								Initial Value
1	1 1 0 0 0 0 0 1								After execution of ROR AL
	→								

In the Rotate shift right, all the bits are shifted towards the right. The least significant bit is shifted to CF as well as rotated to most significant bit, as shown above (in green colour).

CF	AL Register Value								RCL is rotate left with carry;
0	1 0 0 0 0 0 1 1								Initial Value
1	0 0 0 0 0 1 1 0								After execution of RCL AL
	←								

In the Rotate shift left with carry, all the bits are shifted towards the left. The most significant bit is shifted to CF, and the CF is rotated to the least significant bit (shown in blue colour)

CF	AL Register Value								RCR is rotate right with carry;
0	1 0 0 0 0 0 1 1								Initial Value
1	0 1 0 0 0 0 0 1								After execution of RCR AL
	→								

In the Rotate shift right with carry, all the bits are shifted towards the right. The least significant bit is rotated to CF (shown in blue colour), and the CF is shifted to the most significant bit.

Check Your Progress 2

1. Point out the error/ errors in the following 8086 assembly instruction (if any)?
 - a. POPF DX
 - b. MOV BX
 - c. XCHG MEM_BYT1, MEM_BYT2

- d. DAA BL, CL
 - e. DIV AX, CH
2. Compare the different types of shift instructions of 8086 micro-processor.

.....
.....
.....

3. How can you check two operands are equal or not?

.....
.....
.....

13.4.4 Program Execution Transfer Instructions

In general, the execution of instructions of a program is sequential. However, there are certain instructions that results in execution of a different set of instructions. Some of these instructions including call to a procedure, return from a procedure, unconditional and conditional jump instructions etc. All these instructions contain an operand, which is the address of the next instruction which is to be executed as a consequence of execution of this instruction. The conditional jump uses flags register to determine if the jump is to be performed or not. Subroutine call instruction stores the return address. This section explains some of the important program execution transfer instructions.

CALL and RET instructions.

Call and return instructions are used form calling a procedure and once execution of the execution of the procedure is over RET instruction brings the control to the next instruction after the CALL instruction. In 8086 microprocessor there are two types of calls, viz. NEAR call and FAR call. The near call is within the same segment, whereas FAR call is to a different segment. A call instruction has the following basic format:

CALL <address of procedure>

Now, the question is how to recognize, if it is a NEAR or FAR procedure call? This is resolved by the assembler from the declaration of the procedure, which is created as a NEAR or FAR procedure. An example, explaining this is discussed in Unit 15 of this Block. A call to the procedure can be made using the CALL instruction. For example, if the name of a procedure in a separate code segment is procedure1, then the following call instruction will be used:

CALL procedure1 ;

This instruction will cause the execution for following sequence of operations:

1. If, it is a FAR procedure, then, present CS and IP should be saved as return address on the top of the stack, otherwise only IP will be stored on the stack.
SP=SP-2; SS[SP]←CS; // This step will not be required in NEAR procedure
SP=SP-2; SS[SP]←IP;
2. The CS will be loaded with the code segment address of *procedure1* and IP will be loaded with the offset of *procedure1*.
CS= CS of *procedure1*; // This step will not be required in NEAR procedure
IP = Offset of first instruction of *procedure1*;
3. The next instruction as per CS:IP value updated in step 2 will be executed next.

A procedure ends in a return instruction (RET). It causes the called procedure to return to the calling program. The following sequence of actions are performed by the RET instruction.

1. Perform the following actions:..

- ```
CS ← SS[SP] ; SP=SP+2; // NOT performed in NEAR procedure
IP ← SS[SP] ; SP=SP+2;
2. The next instruction as per CS:IP value updated in step 1 will be executed
next.
```

*Jump instructions:*

8086 micro-processor have instructions for unconditional and conditional jump instructions. The unconditional jump can be to NEAR or FAR label. It only requires one operand, which is the address, specified using a Label, of the next instruction to be executed. The format of this instruction is given below:

JMP Label

There are number of unconditional jump instructions. An unconditional jump instruction checks various flag register to determine, if the jump is to be taken or not. One of the most common instruction to set flags prior to conditional jump instruction is the CMP instruction, which has been explained in section 13.4.2. Following table lists some of the conditional jump instructions along with the condition, when the jump will be taken.

| Instruction | Condition if the prior instruction is<br><b>CMP AX, BX or any other arithmetic instruction</b> |
|-------------|------------------------------------------------------------------------------------------------|
| JA/JNBE     | Jump if AX > BX                                                                                |
| JAE/JNB     | Jump if AX >= BX                                                                               |
| JB/JNAE     | Jump if AX < BX                                                                                |
| JBE/JNA     | Jump if AX <= BX                                                                               |
| JC          | Jump if carry flag is set                                                                      |
| JE/JZ       | Jump if AX = BX                                                                                |
| JNC         | Jump if no carry                                                                               |
| JNE/JNZ     | Jump if AX ≠ BX                                                                                |
| JO          | Jump if overflow flag is set                                                                   |
| JNO         | Jump if overflow flag is not set                                                               |
| JP/JPE      | Jump if parity flag is set ; Jump if parity is even                                            |
| JNP/JPO     | Jump if parity flag is not set ; Jump if parity is odd                                         |
| JG/JNLE     | Jump if AX > BX                                                                                |
| JA/JNL      | Jump if AX > BX                                                                                |
| JL/JNGE     | Jump if AX < BX                                                                                |
| JLE/JNG     | Jump if AX <= BX                                                                               |
| JS          | Jump if sign flag is set                                                                       |
| JNS         | Jump if sign flag is not set                                                                   |
| JCXZ        | Jump to specified address if CX =0                                                             |

The instruction JCXZ is a very useful instruction, when CX register is used as a counter.

*Loop instructions:*

A loop instruction (LOOP label) uses *CX register as a counter register*. The label in the loop instruction should be in the range -128 to +127. Prior to a loop instruction, the looping count value should be moved to CX register. The Loop instruction decrements the CX register and checks if CX register has zero value. If CX is not zero, then loop instruction takes the program back to the instruction, which is specified by the label of that instruction. In case, CX is zero then the loop is terminated, i.e., the next instruction after the loop instruction is executed in sequence. 8086 micro-processor has a number of loop instruction, which differ in condition the condition of loop termination. The following table lists some of these instructions, which may be used

later Units. There are many other such instructions for looping, a discussion on them is beyond the scope of this unit.

| Instruction         | Loop Termination                                                        |
|---------------------|-------------------------------------------------------------------------|
| LOOP label          | When CX register is zero.                                               |
| LOOPE/ LOOPZ label  | When a value being checked is unequal OR when CX register becomes zero. |
| LOOPNE/LOOPNZ label | When the value being checked becomes equal or CX register become zero.  |

Example: Let us assume you have byte array of 40h bytes. Write an assembly program segment that check if each of these elements have a value 00F0h.

Solution: Please note the two conditions - the first condition is that each element should be equal to 00F0h and the second condition is loop is to be executed 40h times. Thus, LOOPE instruction would be used, but prior to that you need to set different registers. The program segment for looping is shown below:

```
; Assume that the name of the array is BYTECOST
MOV BX, OFFSET BYTECOST ; This instruction will cause the BX register to
; point at the first element of byte array BYTECOST.
DEC BX ; Decrementing the value of BX register by one.
; This will cause BX to point to one byte prior to
; BYTECOST array. Why is this instruction?
; This is due to specific loop instructions below.
; Initialise the loop counter to size of array
L1: MOV CX,40h ; Move to the next element in the array.
 INC BX ; Compare the array element to 0F0h
 CMP [BX],0F0h ; Loop if the present array element is equal to
 LOOPE L1 ; 0F0h as per CMP instruction and CX is not zero.
```

It may be noted that LOOPE instruction will automatically decrement the value of the counter CX register.

In addition to the program execution control transfer, there are string instructions which are useful for string matching. Such instructions were specially designed for 8086 microprocessor, so that it can perform faster string comparisons. Some of these instructions are discussed in the next section.

### 13.4.5 String Instructions

String based instructions in 8086 were added to allow faster processing of strings based operations. A string is a sequence of ASCII characters in a 8086 microprocessor. Most of the string instructions use a subscript B to indicate that each character in the string is of byte type and a subscript W indicates that each character of the string is of the size of a word (16 bits). Following are the some of the string instructions.

*REP, REPE/REPZ and REPNE/REPNZ:*

These keywords are used before any string instructions to repeat the following instruction to a number of times as specified in CX register. REP prefix repeats the instruction and decrements the CX register by 1, till CX becomes zero. The REPE/REPZ prefix repeats the instruction till either CX becomes zero or ZF becomes 0, whereas REPNE/REPNZ prefix repeats the instruction till either CX becomes zero or ZF becomes 1.

*MOVS/MOVSB/MOVSW instruction:*

This instruction moves data from one byte string to another byte string. This string operator uses several registers implicitly. The source string is assumed to be in data segment, indexed by SI register, whereas the destination string is assumed to be extra data segment indexed by DI register. CX is used as counter register. On transfer of one byte data from sources string to destination, automatically results in increment of SI and DI registers, and decrement of CX register.

Example: Assume that both data segment and extra data segment registers start from segment address 00FFh and a byte string of length 0100h starting at an offset 0400h is to be copied at an offset 0600h. Write the program segment to show this transfer.

; Assuming data segment and extra data segments registered are already initialised.

```
MOV CX, 0100h ; Initialize counter CX
MOV SI, 0400h ; Initialise SI
MOV DI, 0600h ; Initialise DI
REP MOVS ; Will perform transfer till CX is 0.
; SI and DI will be incremented after one byte is transferred
```

*Other string instructions:*

The following table shows other string instructions.

| Instruction          | Purpose                                                                                                                                                                                                                                           |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CMPS/CMPSB/<br>CMPSW | This instruction compares two byte or word strings, use of CX, SI and DI remains the same as MOVS. It is recommended to use REPE in this case.                                                                                                    |
| SCAS/SCASB/<br>SCASW | This instruction compares a string with a value in AL or AX register for a byte or word string respectively. The string to be scanned is assumed to be in extra data segment. This instruction uses CX and DI registers, when REP prefix is used. |
| LODS/LODSB/<br>LODSW | This instruction is used to load a byte or word of a string pointed to by SI register into AL or AX registers respectively.                                                                                                                       |
| STOS/STOSB/<br>STOSW | This instruction is used to store a byte or word from AL or AX registers respectively into a location pointed by DI register.                                                                                                                     |

### 13.4.6 Processor Control Instructions

These instructions are used to change certain parameters that are under the control of the programmer. You can control some of the flags, which may alter the conditional jump and direction of string manipulation. The following table briefly lists some of the most used processor control instructions. You may refer to the further readings for more details on such instructions.

| Instruction | Purpose                                                                                            |
|-------------|----------------------------------------------------------------------------------------------------|
| STC         | This instruction sets the carry flag.                                                              |
| CLC         | This instruction clears the carry flag.                                                            |
| CMC         | This instruction complements or inverts the state of the carry flag.                               |
| STD         | This instruction sets the direction flag (DF=1), so the SI and DI are decremented automatically.   |
| CLD         | This instruction clears the direction flag (DF=0), so the SI and DI are incremented automatically. |

There are many other process control instructions. You may refer to further readings to know more about these instructions.

---

## 13.5 ADDRESSING MODES

---

The 8086 micro-processor supports many operating modes to address the operands. These are – Immediate addressing mode, Register addressing mode, direct addressing mode and Indirect addressing modes. These addressing modes are explained in the following sections.

### 13.5.1 Immediate Addressing Mode

Immediate addressing allows an operand to be part of the instruction. The 8086 assembly language allows you to even use expressions as part of the instructions; however, these expressions should be computable at assembly time to produce a constant value. Some of the examples of immediate source operand are given below:

```
MOV AL, 45h ; move immediate value 45h to AL
MOV AL, 'a' ; move immediate ASCII character value 'a' to AL
MOV AX, 'ba' ; move immediate ASCII characters values to AX register
MOV AL, (5+3)*2 ; move 16 to AL register.
```

### 13.5.2 Register Addressing Mode

A register addressing mode allows any of the register of 8086 to be made as an Operand. However, some special registers cannot be used in every instruction. 8086 microprocessor may allow two register operands in a single instruction. Register operands can be 16 bit registers or 8 bit registers as shown below:

16 bit Register operands: AX,BX,CX,DX,SI,DI,BP,IP,CS,DS,ES,SS

8 bit Register operands: AH, AL, BH, BL, CH, CL, DH, DL

Register operand, in general, allows faster execution of instructions. Some example of instructions using register operands is given below:

```
MOV AL, BL ; Move the content of BL register to AL register
 ; Both the register operands are of 8 bits
MOV DS, AX ; loads data segment register using 16 bit AX register
```

### 13.5.3 Direct Addressing Mode

A direct addressing mode, in general, specifies a memory location as an operand in an instruction. An 8086 instruction can have a maximum of one memory operand. Interestingly, the 8086 memory address is of 16 bits only and contains the offset of an address; therefore, these addresses are called relocatable addresses. If a program is to be reloaded in a different memory segment, it will just require to change the segment register and not the offset. Thus, programs can be relocated to any segment, without changing the instructions. Following are some of the examples of direct addressing mode:

```
MOV CL, LoopCount; loads the content of LoopCount location to CL register.
 ; Segment register used will be data segment (DS)
JMP Loop ; Jump to address specified by the label loop.
 ; Please note that segment register for Loop
 ; would be the code segment.
```

### 13.5.4 Indirect Addressing Mode

In indirect addressing modes, operands use registers to point to locations in memory. So it is actually a register indirect addressing mode. This is a useful mode for handling strings and arrays. For this mode two types of registers are used. These are:

- Base register BX, BP
- Index register SI, DI

BX register contains the offset of the location in Data Segment, whereas BP register points to the base of the stack segment register. The index registers SI and DI also contains offset in the Data Segment and Extra data Segment respectively.

These registers can be combined to create several indirect addressing modes. These are:

*Register indirect:* In this addressing mode the register contains the address of the data. In general, the type of register as stated above determines the segment in which the data is to be accessed. Examples of this mode are:

MOV AL, [DI] ; Move the byte at the memory location ES:DI to AL.  
MOV AL, [BX]; Move the byte at the memory location DS:BX to AL.

*Based indirect:* In this addressing mode a base register and a displacement are added to compute the offset of address of data in the related segment. Example of this mode are:

MOV AL, [BX+2] ; Move the byte at the memory location DS:BX+2 to AL.

*Indexed indirect:* In this addressing mode an index register and a displacement are added to compute the offset of address of data in the related segment. Example of this mode are:

MOV AL, [DI+2] ; Move the byte at the memory location ES:DI+2 to AL.

There are two more such indirect addressing modes, viz. Based Indexed and Based Indexed with displacement, however, they are rarely used and are not explained in this Unit.

### **Check Your Progress 3**

1. Why are CALL and RET statements used?

.....

.....

.....

.....

3. What are the different implicit operations of LOOP instruction?

.....

.....

.....

4. Why do you need the string instructions?

.....

.....

.....

5. Give one example each of each type of addressing mode of 8086 micro-processor.

.....

.....

.....

---

## **13.6 SUMMARY**

---

In this unit, you have gone through the basic architecture of 8086 microprocessor. This architecture was a creative design and used many interesting concepts related to enhancing the speed of instruction processing. First of these is the concept of use of segment registers to reduce 20 bit physical address to a 16 bit offset address, reducing the size of instruction using direct addressing, second faster string processing by using two separate segments to speed up string operations such as matching, third use of pipelining by designing two sections in CPU, fourth use of instruction queue for pre-fetching instructions and so on. 8086 assembly language forms the basis of Intel instruction sets of advanced processors and may help you appreciate the assembly language of those processors.

Some of the key features of this processor include:

- It has 20 bit address bus, therefore, base memory is 1 MB
- It has 16 bit data bus, thus can fetch two bytes simultaneously
- It has four segment registers that along with other pointer registers converts 16 bit offsets to 20 bit physical address.
- It has large number of instructions of different types, which allows writing of powerful assembly programs.

Please refer to the further reading for more details on 8086 assembly language programming.

## 13.7 SOLUTIONS/ANSWERS

### Check Your Progress 1

1. 8086 microprocessor has a Bus interface unit, which is a dedicated unit to compute memory address for reading/ writing a byte or word from/to the memory. It consists of a dedicated adder circuit, which converts 16 bit offset and content of 16 bit segment register to a 20 bit physical address. It also has a 6 byte instruction queue, which can store more than one instruction at a time. The execution unit performs the arithmetic, logical, shift, call, test and many other operations on data. It also contains registers, which store data and temporary results.
2. (a)

|                                       |          |          |          |          |
|---------------------------------------|----------|----------|----------|----------|
| CS (in hexadecimal)                   | 0        | 1        | 1        | 1        |
| Shift left by one Hexadecimal digit   | 0        | 1        | 1        | 0        |
| IP (in hexadecimal)                   | 0        | 0        | 2        | 0        |
| <b>Physical address (Hexadecimal)</b> | <b>0</b> | <b>1</b> | <b>1</b> | <b>0</b> |

(b)

|                                       |          |          |          |          |
|---------------------------------------|----------|----------|----------|----------|
| DS (in hexadecimal)                   | 0        | 2        | 1        | 1        |
| Shift left by one Hexadecimal digit   | 0        | 2        | 1        | 0        |
| BX (in hexadecimal)                   | 0        | 1        | 0        | 0        |
| <b>Physical address (Hexadecimal)</b> | <b>0</b> | <b>2</b> | <b>2</b> | <b>1</b> |

(c)

|                                       |          |          |          |          |
|---------------------------------------|----------|----------|----------|----------|
| SS (in hexadecimal)                   | 4        | 2        | A        | A        |
| Shift left by one Hexadecimal digit   | 4        | 2        | A        | 0        |
| SP (in hexadecimal)                   | 0        | 1        | 2        | 3        |
| <b>Physical address (Hexadecimal)</b> | <b>4</b> | <b>2</b> | <b>B</b> | <b>C</b> |

3. Flag register is used to store all the flag bits, which are generated as a result of last instruction. Some of these flags are sign flag, carry flag, overflow flag etc.  
Flag register cannot be used as a general purpose register.

### **Check Your Progress 2**

1. (a) POPF instruction does not take any explicit operand.  
 (b) Move instruction has a source and destination  
 (c) An instruction cannot have two memory operands in 8086 microprocessor  
 (d) DAA instruction has an implicit operand only. No explicit operand is to specified.  
 (e) In DIV instruction you need to specify one operand only. The other operand is explicit.
2. SHL is shift left instruction and identical to arithmetic shift left instruction.  
 Compare the different types of shift instructions of 8086 micro-processor.  
 However, SHR and SAL differ and different input is added to the left most bit.  
 Rotate instruction ROL and ROR just rotates the word/byte, whereas RCL and RCR also rotate the sign bit. (Please refer to section 13.4.3).
3. Perform test instruction on the operands (please make sure both the operands are not memory operand). If it sets the zero flag, then both the operands are same; otherwise they are different.

### **Check Your Progress 3**

1. CALL statement calls a subroutine, i.e. the next instruction to be executed by the processor should be the first instruction of the subroutine. Since on completion of the subroutine execution the next instruction of the calling program is to execute therefore the return address is stored by the CALL instruction. RET instruction just brings the control back the the next instruction after CALL instruction in the calling program.
2. There are primarily two types of jump instructions: unconditional jump and conditional jumps. The unconditional jump instruction causes a compulsory jump to specified label. There are a number of conditional jump instructions, where a jump is taken if the related condition is fulfilled; else next instruction in sequence is executed.
3. Loop instruction in each iteration decrements CX register, and checks the value of CX. In case it is not zero, you go back to the Label from where the loop started. However, if the CX register is zero, the next instruction in sequence is executed.
4. String instructions in 8086 microprocessor are specially designed for efficient execution of string operations. For example, to match two strings, one string each be put in DS and ES with DS:SI pointing to first string and ES:DI pointing to second string. String length is put in CX register. The string matching instruction on using REPE command will compare the first byte and will increment SI and DI; and decrement CX. Thus, you do not need to write lengthy program for string matching, which includes all the operation as given above.
5. Immediate Operand

MOV AL, (9+7)\*2 ; move 32 to AL register.

Register Addressing

MOV AL, DL ; move DL to AL register.

Direct Addressing

MOV AL, X ; move content of byte location X to AL register.

Register Addressing

MOV AH, [BX] ; move content of location, whose address is ; DS:BX to AL register.