# UNIT 1 MICROPROCESSOR ARCHITECTURE

**Structure** **Page No.**



## 1.0 INTRODUCTION

In the previous blocks of this course, we have discussed concepts relating to CPU organization, register set, instruction set, addressing modes with a few examples. Let us look at one microprocessor architecture in regard of all the above concepts. We have selected one of the simplest processors 8086, for this purpose. Although the processor technology is old, all the concepts are valid for higher end Intel processor. Therefore, in this unit, we will discuss the 8086 microprocessor in some detail.

We have started the discussion of the basic microcomputer architecture. This discussion is followed by the details on the components of CPU of the 8086 microprocessor. Then we have discussed the register organization for this processor. We have also discussed the instruction set and addressing modes for this processor. Thus, this unit presents exhaustive details of the 8086 microprocessor. These details will then be used in Assembly Programming.

## 1.1 OBJECTIVES

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

- describe the features of the 8086 microprocessor;
- list various components of the 8086 microprocessor; and
- identify the instruction set and the addressing modes of the 8086 microprocessor.

## 1.2 MICROCOMPUTER ARCHITECTURE

The word micro is used in microscopes, microphones, microwaves, microprocessors, microcomputers, microprogramming, microcodes etc. It means small. A

microprocessor is an example of VLSI bringing the whole processor to a single small chip. With the popularity of distributed processing, the emphasis has shifted from the single mainframe system to independently working workstations or functioning units with their own CPU, RAM, ROM and a magnetic or optical disk memory. Thus, the advent of the microprocessor has transformed the mainframe environment to a distributed platform.

Let us recapitulate the basic components of a microprocessor:

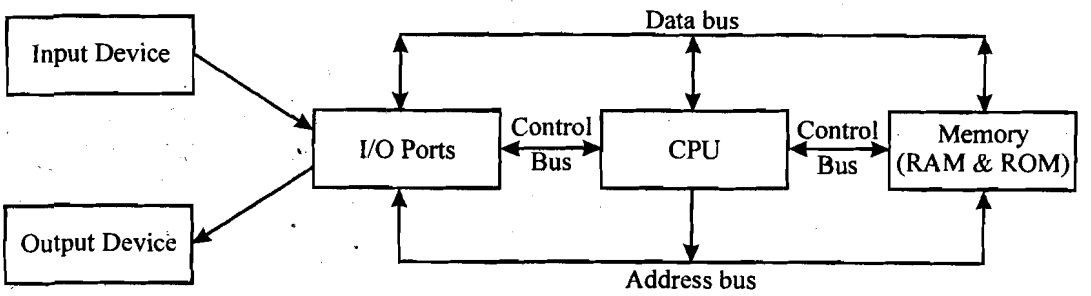



**Figure 1: Components of a Microcomputer**

Please note the following in the above figure:

- ROM stores the boot program.
- The path from CPU to devices is through Buses. But what would be the size of these Buses?

**Bus Sizes**

1. The Address bus: 8085 microprocessor has 16 bit lines. Thus, it can access up to $2^{16}$ = 64K Bytes. The address bus of 8086 microprocessor has a 20 bits address bus. Thus it can access upto $2^{20}$ = 1M Byte size of RAM directly.

2. Data bus is the number of bits that can be transferred simultaneously. It is 16 bits in 8086.

**Microprocessors**

The microprocessor is a complete CPU on a single chip. The main advantages of the microprocessor are:

- compact but powerful;
- can be microprogrammed for user's needs;
- easily programmable and maintainable due to small size; and
- useful in distributed applications.

A microprocessor must demonstrate:

- More throughput
- More addressing capability
- Powerful addressing modes
- Powerful instruction set
- Faster operation through pipelining
- Virtual memory management.

However, RISC machine do not agree with above principles.

Some of the most commercially available microprocessors are: Pentium, Xeon, G4 etc.

The assembly language for more advanced chips subsumes the simplest 8086/ 8088 assembly language. Therefore, we will confine our discussions to Intel 8086/8088 assembly language. You must refer to the further readings for more details on assembly language of Pentium, G4 and other processors.

All microprocessors execute a continuous loop of fetch and execute cycles.

```
while (1)
{
        fetch (instruction);
        execute (using date);
}
```

## 1.3 STRUCTURE OF 8086 CPU

The 8086 microprocessor consists of two independent units:

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

Please refer to Figure 2.

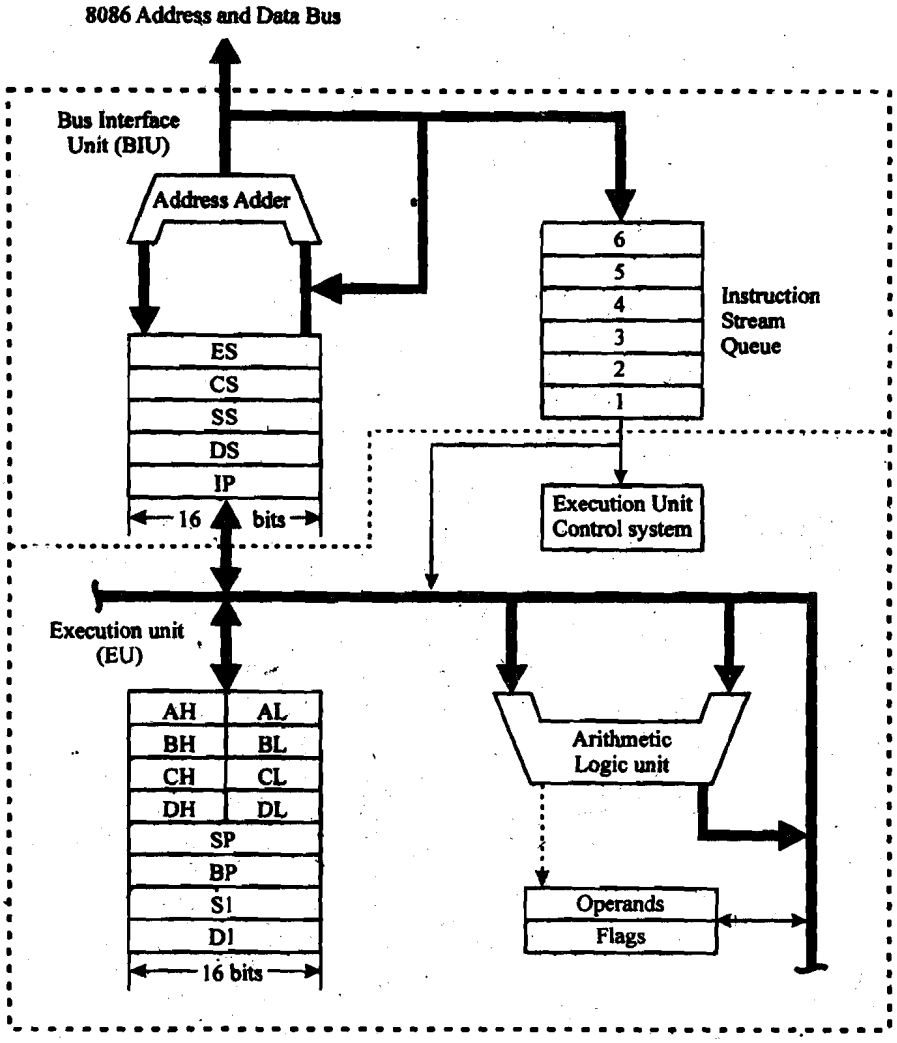



**Figure 2: The CPU of INTEL 8086 Microprocessor**

The word independent implies that these two units can function parallel to each other. In other words they may be considered as two stages of the instruction pipeline.

### 1.3.1 The Bus Interface Unit

The BIU (Bus Interface Unit) primarily interacts with the system bus. It performs almost all the activities relating to fetch cycle such as:

- Calculating the physical address of the next instruction
- Fetching the instruction
- Reading or writing data memory or I/O port from memory or Input/ Output.

The instruction/ data is then passed to the execution unit. This BIU consists of:

**(a) The Instruction Queue**

The instruction queue is used to store the instruction "bytes" fetched. Please note two points here: that it is (1) A Byte (2) Queue. This is used to store information in byte form, with the underlying queue data structure. The advantage of this queue would only be if the next expected instructions are fetched in advance, thus, allowing a pipeline of fetch and execute cycles.

**(b) The Segment Registers**

These are very important registers of the CPU. Why? We will answer this later. In 8086 microprocessor, the memory is a byte organized, that is a memory address is byte address. However, the number of bits fetched is 16 at a time. The segment registers are used to calculate the address of memory location along with other registers. A segment register is 16 bits long.

The BIU contains four sixteen-bit registers, viz., the CS: Code Segment, the DS: Data Segment, the SS: Stack Segment, and the ES: Extra Segment. But what is the need of the segments: Segments logically divide a program into logical entities of Code, Data and Stack each having a specific size of 64 K. The segment register holds the upper 16 bits of the starting address of a logical group of memory, called the segment. But what are the advantages of using segments? The main advantages of using segments are:

- Logical division of program, thus enhancing the overall possible memory use and minimise wastage.
- The addresses that need to be used in programs are relocatable as they are the offsets. Thus, the segmentation supports relocatability.
- Although the size of address, is 20 bits, yet only the maximum segment size, that is 16 bits, needs to be kept in instruction, thus, reducing instruction length.

The 8086 microprocessor uses overlapping segments configuration. The typical memory organization for the 8086 microprocessor may be as per the following figure.

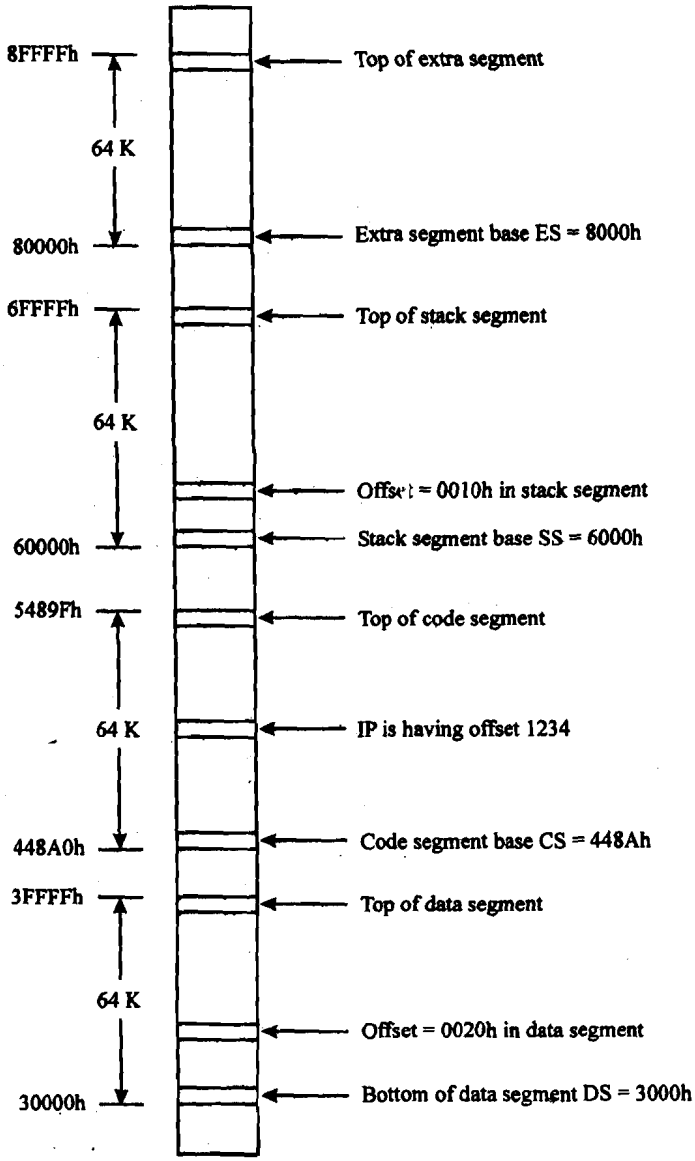



**Figure 3: Logical Organisation of Memory in INTEL 8086 Microprocessor**

Although the size of each segment can be 64K, as they are overlapping segments we can create variable size of segments, with maximum as 64K. Each segment has a specific function. 8086 supports the following segments:

As per model of assembly program, it can have more than one of any type of segments. However, at a time only four segments one of each type, can be active.

The 8086 supports 20 address lines, thus supports 20 bit addresses. However, all the registers including segment registers are of only 16 bits. So how may this mapping of 20 bits to 16 bits be performed?

Let us take a simple mapping procedure:

The top four hex digits of initial physical address constitute segment address.

You can add offset of 16 bits (4 Hex digits) from 0000h to FFFFh to it . Thus, a typical segment which starts at a physical address 10000h will range from 10000h to 1FFFFh, The segment register for this segment will contain 1000H and offset will

range from 0000h to FFFFh. But, how will the segment address and offset be added to calculate physical address? Let us explain using the following examples:

**Example 1 (In the Figure above)**

The value of the stack segment register (SS) = 6000h
The value of the stack pointer (SP) which is Offset = 0010h

Thus, Physical address of the top of the stack is:

| | | | | | | |
|---|---|---|---|---|---|---|
| SS | 6 | 0 | 0 | 0 | 0 | Implied zero |
| SP + | 0 | 0 | 1 | 0 | | |
| Physical Address | 6 | 0 | 0 | 1 | 0 | |

This calculation can be expressed as:
Physical address = SS (hex) × 16 + SP (hex)

**Example 2**

The offset of the data byte = 0020h
The value of the data segment register (DS) = 3000h
Physical address of the data byte

| | | | | | | |
|---|---|---|---|---|---|---|
| DS | 3 | 0 | 0 | 0 | 0 | Implied Zero |
| Offset + | 0 | 0 | 2 | 0 | | |
| Physical Address | 3 | 0 | 0 | 2 | 0 | |

This calculation can be expressed as physical address = DS (Hex) × 16 + Data byte offset (hex).

**Example 3**

The value of the Instruction Pointer, holding address of the instruction = 1234h
The value of the code segment register (CS) = 448Ah
Physical address of the instruction

| | | | | | | |
|---|---|---|---|---|---|---|
| CS | 4 | 4 | 8 | A | 0 | ImpliedZero |
| + | 1 | 2 | 3 | 4 | | |
| IP | | | | | | |
| Physical Address | 4 | 5 | A | 0 | 4 | |

Physical Address = CS (Hex) × 16 + IP

**(c) Instruction Pointer**

The instruction pointer points to the offset of the current instruction in the code segment. It is used for calculating the address of instruction as shown above.

### 1.3.2 Execution Unit (EU)

Execution unit performs all the ALU operations. The execution unit of 8086 is of 16 bits. It also contains the control unit, which instructs bus interface unit about which memory location to access, and what to do with the data. Control unit also performs decoding and execution of the instructions. The EU consists of the following:

**(a) Control Circuitry, Instruction Decoder and ALU**

The 8086 control unit is primarily micro-programmed control. In addition it has an instruction decoder, which translates an instruction into sequence of micro operations. The ALU performs the required operations under the control of CU which issues the necessary timing and control sequences.

**(b) Registers**

All CPUs have a defined number of operational registers. 8086 has several general purpose and special purpose registers. We will discuss these registers in the following sections.

## 1.4 REGISTER SET OF 8086

The 8086 registers have five groups of registers. These groupings are done on the basis of the main functions of the registers. These groups are:

**General Purpose Register**

8086 microprocessors have four general purpose registers namely, AX, BX, CX, DX. All these registers are 16 – bit registers. However, each register can be used as two general-purpose byte registers also. These byte registers are named AH and AL for AX, BH and BL for BX, CH and CL for CX, and DH and DL for DX. The H in register name represents higher byte while L represents lower byte of the 16 bits registers. These registers are primarily used for general computation purposes. However, in certain instruction executions they acquire a special meaning.

**AX** register is also known as accumulator. Some of the instructions like divide, rotate, shift etc. require one of the operands to be available in the accumulator. Thus, in such instructions, the value of AX should be suitably set prior to the instruction.

**BX** register is mainly used as a base register. It contains the starting base location of a memory region within a data segment.

**CX** register is a defined counter. It is used in loop instruction to store loop counter.

**DX** register is used to contain I/O port address for I/O instruction.

You will experience their usage in various assembly programs discussed later.

**Segment Registers**

Segment Registers are used for calculating the physical address of the instruction or memory. Segment registers cannot be used as byte registers.

**Pointer and Index Registers**

The 8086 microprocessor has three pointer and index registers. Each of these registers is of 16 bit and cannot be accessed byte wise. These are Base Pointer (BP), Source Index (S1) and Destination Index (DI). Although they can be used as general purpose registers, their main objective is to contain indexes. BP is used in stack segment, SI in Data segment and DI in Extra Data segment.

### Special Registers

A Last in First Out (LIFO) stack is a data structure used for parameter passing, return address storage etc. 8086 stack is 64K bytes. Base of the stack is pointed to by the stack segment (SS) register while the offset or top of the stack is stored in Stack Pointer (SP) register. Please note that although the memory in 8086 has byte addresses, stack is a word stack, which is any push operation will occupy two bytes.

### Flags Register

A flag represents a condition code that is 0 or 1. Thus, it can be represented using a flip- flop. 8086 employs a 16-bit flag register containing nine flags. The following table shows the flags of 8086.

| Flags | Meaning | Comments |
|---|---|---|
| Conditional Flags represent result of last arithmetic or logical instruction executed. Conditional flags are set by some condition generated as a result of the last mathematical or logical instruction executed. The conditional flags are: | | |
| CF | Carry Flag | 1 if there is a carry bit |
| PF | Parity Flag | 1 on even parity 0 on odd parity |
| AF | Auxiliary Flag | Set (1) if auxiliary carry for BCD occurs |
| ZF | Zero Flag | Set if result is equal to zero |
| SF | Sign Flag | Indicates the sign of the result (1 for minus, 0 for plus) |
| OF | Overflow Flag | set whenever there is an overflow of the result |
| Control flags, which are set or reset deliberately to control the operations of the execution unit. The control flags of 8086 are as follows: | | |
| TF | Single step trap flag | Used for single stepping through the program |
| IF | Interrupt Enable flag | Used to allow/inhibit the interruption of the program |
| DF | String direction flag | Used with string instruction. |

## ☞ Check Your Progress 1

1. What is the purpose of the queue in the bus interface unit of 8086 microprocessors?

   .....

   .....

   .....

2. Find out the physical addresses for the following segment register: offset

   (a) SS:SP = 0100h:0020h
   (b) DS:BX = 0200h:0100h
   (c) CS:IP = 4200h:0123h

3. State True or False. | T | F |

(a) BX register is used as an index register in a data segment. ☐

(b) CX register is assumed to work like a counter. ☐

(c) The Source Index (SI) and Destination Index(DI) registers in 8086 can also be used as general registers. ☐

(d) Trag Flag (TR) is a conditional flag. ☐

## 1.5 INSTRUCTION SET OF 8086

After discussing the basic organization of the 8086 micro-processor, let us now provide an overview of various instructions available in the 8086 microprocessor. The instruction set is presented in the tabular form. An assembly language instruction in the 8086 includes the following:

**Label: Op-code Operand(s); Comment**

For example, to add the content of AL and BL registers to get the result in AL, we use the following assembly instruction.

**NEXT: ADD AL,BL ; AL ← AL + BL**

Please note that NEXT is the label field. It is giving an identity to the statement. It is an optional field, and is used when an instruction is to be executed again through a LOOP or GO TO. ADD is symbolic op-code, for addition operation. AL and BL are the two operands of the instructions. Please note that the number of operands is dependent upon the instructions. 8086 instructions can have zero, one or two operands. An operand in 8086 can be:

1. A register
2. A memory location
3. A constant called literal
4. A label.

We will discuss the addressing modes of these operands in section 1.6.

Comments in 8086 assembly start with a semicolon, and end with a new line. A long comment can be extended to more than one line by putting a semicolon at the beginning of each line. Comments are purely optional, however recommended as they provide program documentation. In the next few sections we look at the instruction set of the 8086 microprocessor. These instructions are grouped according to their functionality.

### 1.5.1 Data Transfer Instructions

These instructions are used to transfer data from a source operand to a destination operand. The source operand in most of the cases remains unchanged. The operand can be a literal, a memory location, a register, or even an I/O port address, as the case may be. Let us discuss these instructions with the following table:

| MNEMONIC | DESCRIPTION | EXAMPLE |
|---|---|---|
| MOV des, src | des ← src; Both the operands should be byte or word. **src** operand can be register, memory location or an immediate operand **des** can be register or memory operand. Restriction: Both source and destination cannot be memory operands at the same time. | MOV CX,037AH<br>; CX register is initialized<br>; with immediate value<br>; 037AH.<br><br>MOV AX,BX<br>; AX←BX |

| | | |
|---|---|---|
| PUSH operand | Pushes the operand into a stack.<br>SP ← SP – 2;<br>value [TOS] ← operand.<br>Initialise stack segment register, and the stack pointer properly before using this instruction. No flags are effected by this instruction. The operand can be a general purpose register, a segment register, or a memory location. Please note it is a word stack and memory address is a byte address, thus, you decrement by 2. Also you decrement as SP is initialised to maximum offset and condition of stackful is a zero offset (so it is a reversed stack) | PUSH BX<br>; decrement stack pointer<br>; by; two, and copy BX to<br>; stack.<br>; decrement stack pointer<br>; by two, and copy<br>; BX to stack |
| POP des | POP a word from stack. The **des** can be a general-purpose register, a segment register (except for CS register), or a memory location. Steps are:<br>des← value [TOS]<br>SP ← SP + 2 | POP AX<br>; Copy content for top<br>; of stack to AX. |
| XCHG des, src | Used to exchange bytes or words of src and des. It requires at least one of the operands to be a register operand. The other can be a register or memory operand. Thus, the instruction cannot exchange two memory locations directly. Both the operands should be either byte type or word type. The segment registers cannot be used as operands for this instruction. | XCHG DX,AX<br>; Exchange word in DX<br>; with word in AX |
| XLAT | Translate a byte in AL using a table stored in the memory. The instruction replaces the AL register with a byte from the lookup table. This instruction is a complex instruction. | Example is available in Unit 3. |
| IN accumulator, port address | It transfers a byte or word from specified port to accumulator register. In case an 8-bit port is supplied as an operand then the data byte read from that part will be transferred to AL register. If a 16-bit port is read then the AX will get 16 bit word that was read. The port address can be an immediate operand, or contained in DX register. This instruction does not change any flags. | IN AL,028h<br>; read a byte from port<br>; 028h to AL register |
| OUT port address, Accumulator | It transfers a byte or word from accumulator register to specified port. This instruction is used to output on devices like the monitor or the printer. | |
| LEA register, source | Load "effective address" (refer to this term in block 2, Unit 1 in addressing modes) of operand into specified 16 – bit register. Since, an address is an offset in a segment and maximum can | LEA BX, PRICES<br>; Assume PRICES is<br>; an array in the data<br>; segment. The<br>; instruction loads the |

| | | |
|---|---|---|
| | be of 16 bits, therefore, the register can only be a 16-bit register. LEA instruction does not change any flags. The instruction is very useful for array processing. | ; offset of the first byte of ; PRICES directly into ; the BX register. |
| LDS des-reg | It loads data segment register and other specified register by using consecutive memory locations. | LDS SI, DATA ; DS←content of memory ; location DATA & ; DATA + 1 ; SI ← content of ; memory locations ; DATA + 2 & DATA + ; 3 |
| LES des-reg | It loads ES register and other specified register by using consecutive memory locations. This instruction is used exactly like the LDS except in this case ES & other specified registers are initialized. | |
| LAHF | Copies the lower byte of flag register to AH. The instruction does not change any flags and has no operands. | |
| SAHF | Copies the value of AH register to low byte of flag register. This instruction is just the opposite of LAHF instruction. This instruction has no operands. | |
| PUSHF | Pushes flag register to top of stack. SP ← SP – 2; stack [SP] ← Flag Register. | |
| POPF | Pops the stack top to Flag register. Flag register ← stack [SP] SP ← SP + 2 | |

### 1.5.2 Arithmetic Instructions

| MNEMONIC | DESCRIPTION | EXAMPLE |
|---|---|---|
| ADD | Adds byte to byte, or word to word. The source may be an immediate operand, a register or a memory location. The rules for operands are the same as that of MOV instruction. To add a byte to a word, first copy the byte to a word location, then fill up the upper byte of the word with zeros. This instruction effects the following flags: AF, CF, OF, PF, SF, ZF. | ADD AL,74H ; Add the number 74H to ; AL register, and store the ; result back in AL ADD DX,BX ; Add the contents of DX to ; BX and store the result in ; DX, BX remains ; unaffected. |
| ADC des, src | Add byte + byte + carry flag, or word + word + carry flag. It adds the two operands with the carry flag. Rest all the details are the same as that of ADD instruction. | |
| INC des | It increments specified byte or word operand by one. The operand can be a register or a memory location. It can effect AF, SF, ZF, PF, and OF flags. It does not affect the carry flag, that is, if you increment a byte operand | INC BX ; Add 1 to the contents of ; BX register INC BL ; Add 1 to the contents of ; BL register |

| | | |
|---|---|---|
| | having 0FFH, then it will result in 0 value in register and no carry flag. | |
| AAA | ASCII adjusts after addition. The data entered from the terminal is usually in ASCII format. In ASCII 0-9 are represented by codes 30-39. This instruction allows you to add the ASCII codes instead of first converting them to decimal digit using masking of upper nibble. AAA instruction is then used to ensure that the result is the correct unpacked BCD. | ADD AL,BL<br>; AL=00110101, ASCII 05<br>; BL=00111001, ASCII 09<br>; after addition<br>; AL = 01101110, that is,<br>; 6EH- incorrect<br>; temporary result<br>AAA<br>; AL = 00000100.<br>; Unpacked BCD for 04<br>; carry = 1, indicates<br>; the result is 14 |
| DAA | Decimal (BCD) adjust after addition. This is used to make sure that the result of adding two packed BCD numbers is adjusted to be a correct BCD number. DAA only works on AL register. | ; AL = 0101 1001 (59<br>; BCD)<br>; BL = 0011 0101 (35<br>; BCD)<br>ADD AL, BL<br>; AL = 10001101 or<br>; 8Eh (incorrect BCD)<br>DAA<br>; AL = 1001 0100<br>; ≡ 94 BCD : Correct. |
| SUB des, src | Subtract byte from byte, or word from word. (des ← des – src). For subtraction the carry flag functions as a borrow flag, that is, if the number in the source is greater than the number in the destination, the borrow flag is to set 1. Other details are equivalent to that of the ADD instruction. | SUB AX, 3427h<br>; Subtract 3427h from AX<br>; register, and store the<br>; result back in AX |
| SBB des, src | Subtract operands involving previous carry if any. The instruction is similar to SUB, except that it allows us to subtract two multibyte numbers, because any borrow produced by subtracting less-significant byte can be included in the result using this instruction. | SBB AL,CH<br>; subtract the contents<br>; of CH and CF from AL<br>; and store the result<br>; back in AL. |
| DEC src | Decrement specified byte or specified word by one. Rules regarding the operands and the flags that are affected are same as INC instruction. Please note that if the contents of the operand is equal to zero then after decrementing the contents it becomes 0FFH or 0FFFFH, as the case may be. The carry flag in this case is not affected. | DEC BP<br>; Decrement the contents<br>; of BP<br>; register by one. |
| NEG src | Negate - creates 2's complement of a given number, this changes the sign of a number. However, please note that if you apply this instruction on operand having value –128 (byte operand) or –32768 (word operand) it will result in overflow condition. The overflow (OF) flag will be set to | NEG AL<br>; Replace the number in<br>; AL with it's 2's<br>; complement |

| | | |
|---|---|---|
| | indicate that operation could not be done. | |
| CMP des,src | It compares two specified byte operands or two specified word operands. The source and destination operands can be an immediate number, a register or a memory location. But, both the operands cannot be memory locations at the same time. *The comparison is done simply by internally subtracting the source operand from the destination operand.* The value of source and the destination, operand is not changed, but the flags are set to indicate the results of the comparison. | CMP CX,BX<br>; Compare the CX register<br>; with the BX register<br>; In the example above, the ;<br>CF, ZF, and the SF flags<br>; will be set as follows.<br>; CX=BX 0 1 0; result of<br>; subtraction is zero<br>; CX>BX 0 0 0; no borrow ;<br>required therefore, CF=0<br>; CX<BX 1 0 1<br>; subtraction require<br>; borrow, so CF=1 |
| AAS | ASCII adjust after subtraction. This instruction is similar to AAA (ASCII adjust after addition) instruction. The AAS instruction works on the AL register only. It updates the AF and CF flags, but the OF, PF, SF and the ZF flags remain undefined. | ; AL = 0011 0101 ASCII 5<br>; BL = 0011 1001 ASCII 9<br>SUB AL,BL<br>; (5-9) result:<br>; AL= 1111 1100 = - 4 in<br>; 2's complement, CF = 1<br>AAS ;result:<br>; AL = 0000 0100 =<br>; BCD 04,<br>; CF = 1 borrow needed. |
| DAS | Decimal adjust after subtraction. This instruction is used after subtracting two packed BCD numbers to make sure the result is the packed BCD. DAS only works on the AL register. The DAS instruction updates the AF, CF, SF, PF and ZF flags. The overflow (OF) is undefined after DAS. | ; AL=86 BCD<br>; BH=57 BCD<br>SUB AL,BH<br>; AL=2Fh, CF =0<br>DAS<br>; Results in AL = 29 BCD |
| MUL src | This is an unsigned multiplication instruction that multiplies two bytes to produce a word operand or two words to produce a double word such as:<br>AX ← AL* src (byte multiplication src is also byte)<br>DX or AX ← AX * src (word multiplication is two word).<br>This instruction assumes one of the operand in AL (byte) or AX (word): the src operand can be register or memory operand. If the most significant word of the result is zero then, the CF and the OF flags are both made zero. The AF, SF, PF, ZF flags are not defined after the MUL instruction. If you want to multiply a byte with a word, then first convert byte to a word operand. | MOV AX,05; AX=05<br>MOV CX,02; CX=02<br>MUL CX<br>; results in DX=0<br>; AX=0Ah |
| AAM | ASCII adjust after multiplication. Please note that two ASCII numbers cannot be multiplied directly. To multiply first convert the ASCII | ; AL=0000 0101 unpacked<br>; BCD 05<br>; BH=0000 1001 unpacked ;<br>BCD 09 |

| | | |
|---|---|---|
| | number to numeric digits by masking off the upper nibble of each byte. This leaves unpacked BCD in the register. AAM instruction is used to adjust the product to two unpacked BCD digits in AX after the multiplication has been performed. AAM defined by the instruction while the CF, OF and the AF flags are left undefined. | MUL BH<br>; AX=AL * BH=002Dh<br>AAM<br>; AX=00000100 00000101 ;<br>BCD 45 : Correct result |
| DIV src | This instruction divides unsigned word by byte, or unsigned double word by word. For dividing a word by a byte, the word is stored in AX register, divisor the src operand and the result is obtained in AH : remainder AL: quotient. It can be represented as:<br>AH: Remainder<br>AL: Quotient } ← AX/ src<br>Similarly for double word division by a word we have<br>DX: Remainder<br>AX: Quotient } ← DX:AX/ src<br>A division by zero result in run time error. The divisor src can be either in a register or a memory operand. | ; AX = 37D7h = 14295<br>; decimal<br>; BH = 97h = 151 decimal<br>DIV BH<br>; AX / BH quotient<br>; AL = 5Eh = 94<br>; decimal RemainderAH = ;<br>65h = 101<br>; decimal |
| IDIV | Divide signed word by byte or signed double word by word. For this division the operand requirement, the general format of the instruction etc. are all same as the DIV instruction. IDIV instruction leaves all flags undefined. | ; AL = 11001010 = -26h =<br>; - 38 decimal<br>; CH = 00000011 = + 3h =<br>; 3 decimal<br>; According to the operand<br>; rules to divide by a byte<br>; the number should be<br>; present in a word register, ;<br>i.e. AX. So, first convert<br>; the operand in AL to word<br>; operand. This can be done ;<br>by sign extending the<br>; AL register,<br>; this makes AX<br>; 11111111 11001010.<br>; (Sign extension can also<br>; be done with the help of<br>; an instruction, discussed<br>; later)<br>IDIV CH<br>; AX/CH<br>; AL = 11110100 = - 0CH ;<br>= -12 Decimal<br>; AH = 11111110 = -02H = ; -<br>02 Decimal<br>; Although the quotient is<br>; actually closer to -13<br>; (-12.66667) than -12, but<br>; 8086 truncates the result<br>; to give -12. |
| AAD | ASCII adjust after division. The BCD numbers are first unpacked, by | ; AX= 0607 unpacked<br>; BCD for 6 |

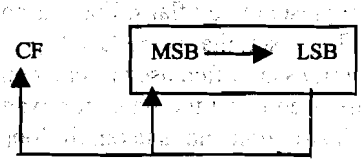
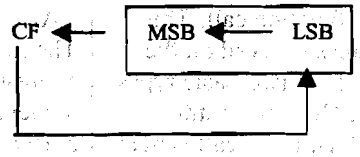
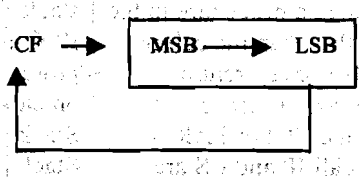
| | | |
|---|---|---|
| | masking off the upper nibble of each byte. Then ADD instruction is used to convert the unpacked BCD digits in AL and AH registers to adjust them to equivalent binary prior to division. Such division will result in unpacked BCD quotient and remainder. The PF, SF, ZF flags are updated, while the AF, CF, and the OF flags are left undefined. | ; and 7 CH = 09h<br>AAD<br>; adjust to binary before<br>; division AX= 0043 =<br>; 043h = 67 Decimal<br>DIV CH<br>; Divide AX by unpacked<br>; BCD in CH<br>; AL = 07 unpacked BCD<br>; AH = 04 unpacked BCD<br>; PF = SF = ZF = 0 |
| CBW | Fill upper-byte or word with copies of sign bit of lower bit. This is called sign extension of byte to word. This instruction does not change any flags. This operation is done with AL register in the result being stored in AX. | ; AL = 10011011 = -155<br>; decimal AH = 00000000<br>CBW ;convert signed<br>; byte in AL to signed<br>; word in AX = 11111111<br>; 10011011 = -155 decimal |
| CWD | Fill upper word or double word with sign bit of lower word. This instruction is an extension of the previous instruction. This instruction results in sign extension of AX register to DX:AX double word. | ; DX : 0000 0000 0000 0000<br>; AX : 1111 0000 0101 0001<br>CWD<br>; DX:AX = 1111 1111 1111 1111:<br>; 1111 0000 0101 0001 |

### 1.5.3 Bit Manipulation Instructions

These instructions are used at the bit level. These instructions can be used for testing a zero bit, set or reset a bit and to shift bits across registers. Let us look into some such basic instructions.

| MNEMONIC | DESCRIPTION | EXAMPLE |
|---|---|---|
| NOT des | Complements each bit to produce 1's complement of the specified byte or word operand. The operand can be a register or a memory operand. | ; BX = 0011 1010 0001 0000<br>NOT BX<br>; BX = 1100 0101 1110 1111 |
| AND des, src | Bitwise AND of two byte or word operands. The result is des ← des AND src. The source can be an immediate operand a register, or a memory operand. The destination can be a register or a memory operand. Both operands cannot be memory operands at the same time. The CF and the OF flags are both zero after the AND operation. PF, SF and ZF area updated, Afis left undefined. | ; BH = 0011 1010 before<br>AND BH, 0Fh<br>; BH = 0000 1010<br>; after the AND operation |
| OR des, src | OR each corresponding bits of the byte or word operands. The other operands rules are same as AND. des ← des OR src | ; BH = 0011 1010 before<br>OR BH, 0Fh<br>; BH = 0011 1111 after |
| XOR des,src | XOR each corresponding bit in a byte or word operands rules are two same as AND and OR. des ← Des + src | ; BX = 00111101 01101001<br>; CX = 00000000 11111111<br>XOR BX,CX<br>; BX=0011110110010110<br>; Please note, that the bits in<br>; the lower byte are inverted. |

| | | |
|---|---|---|
| TEST des, src | AND the operands to update flags, but donot change operands value. It can be used to set and test conditions.CF and OF are both set to zero, PF, SF and ZF are all updated, AF is left undefined after the operation. | ; AL = 0101 0001<br>TEST AL, 80h.<br>; This instruction would<br>; test if the MSB bit of the AL<br>; register is zero or one. After<br>; the TEST operation ZF will<br>; be set to 1 if the MSB of AL<br>; is zero. |
| SHL/SAL des, count | Shift bits of word or byte left, by count. It puts zero(s) in LSB(s). MSB is shifted into the carry flag. If more than one bits are shifted left, then the CF gets the most recently moved MSB. If the number of bits desired to be shifted is only 1, then the immediate number. 1 can be written as one of the operands. However, if the number of bits desired to be shifted is more than one, then the second operand is put in CL register. | SAL BX, 01<br>; if CF = 0<br>; BX = 1000 1001<br>; result : CF = 1<br>; BX = 0001 0010 |
| SHR des, count | It shifts bits of a byte or word to register put zero in MSB. LSB is moved into CF. | SHR BX,01<br>; if CF = 0<br>; BX = 1000 1001<br>; result: CF = 1<br>; BX = 0100 0100<br>MOV CL, 02<br>SHR BX, CL<br>; with same BX, the<br>; result would be<br>; CF = 0<br>; BX = 0010 0100 |
| SAR des, count | Shift bits of word or byte right, but it retains the value of new MSB to that of old MSB. This is also called arithmetic shift operation, as it does not change the MSB, which is sign bit of a number. | ; AL=0001 1101 = +29<br>; decimal, CF = 0<br>SAR AL, 01<br>; AL = 0000 1110 = +14<br>; decimal, CF = 1<br>; OF = PF = SF = ZF = 0<br>; BH = 1111 0011 = -13<br>; decimal<br>SAR BH,01<br>; BH = 1111 1001 = -7<br>; decimal, CF =1<br>; OF = ZF = 0 ; PF = SF = 1 |
| ROL des, count | Rotate bits of word or byte left, MSB is transferred to LSB and also to CF. Diagrammatically, it can be represented as:<br>  <br><br>The operation is called rotate as it circulates bits. The operands can be register or memory operand. | |
| ROR des, count | Rotate bits of word or byte right, | ; CF = 0, |

| | | |
|---|---|---|
| | LSB is transferred to MSB and also to CF. The same can be represented diagrammatically as follows:   | ; BX = 0011 1011 0111 0101<br>ROR BX, 1<br>; results ; CF = 1,<br>; BX = 1001 1101 1011 1010 |
| RCL des, count | Rotate bits of words or byte left, MSB to CF and CF to LSB. The operation is circular and involves carry flag in rotation.   | |
| RCR des, count | Rotate bits of word or byte right, LSB to CF and CF to MSB. This instruction rotates left.   | |

## Check Your Progress 2

1. Point out the error/ errors in the following 8086 assembly instruction (if any)?

   a. PUSHF AX
   b. MOV AX, BX
   c. XCHG MEM_WORD1, MEM_WORD2
   d. AAA BL, CL
   e. IDIV AX, CH

2. State True or False in the context of 8086 assembly language. | T | F |

   (a) LEA and MOV instruction serve the same purpose. The only difference between the two is the type of operands they take. ☐

   (b) NEG instruction produces 1's complement of a number. ☐

   (c) MUL instruction assumes one of the operands to be present in the AL or AX register. ☐

   (d) TEST instruction performs an OR operation, but does not change the value of operands. ☐

   (e) Suppose AL contains 0110 0101 and CF is set, then instructions ROL AL and RCL AL will produce the same results. ☐

### 1.5.4 Program Execution Transfer Instructions

These instructions are the ones that causes change in the sequence of execution of instruction. This change can be through a condition or sometimes may be unconditional. The conditions are represented by flags. For example, an instruction may be jump to an address if zero flag is set, that is the last ALU operation has resulted in zero value. These instructions are often used after a compare instruction, or some arithmetic instructions that are used to set the flags, for example, ADD or SUB. LOOP is also a conditional branch instruction and is taken till loop variable is below a certain count.

Please note that a "/" is used to separate two mnemonics which represent the same instruction.

| MNEMONIC | DESCRIPTION | EXAMPLE |
|---|---|---|
| CALL proc 1 | This function results in a procedure/ function call. The return address is saved on the stack. There are two basic types of CALLS. NEAR or Intra-Segment calls: if the call is made to a procedure in the same segment as the calling program. FAR or Inter segment call: if the call is made to a procedure in the segment, other than the calling program. The saved return address for NEAR procedure call is just the IP. For FAR Procedure call IP and CS are saved as return address.<br><br>A procedure can also be called indirectly, by first initializing some 16-bit register, or some other memory location with the new addresses as follows. | CALL procl<br>CALL proc2<br>The new instruction address is determined by name declaration procl is a near procedure, thus, only IP is involved. proc2 involves new CS: IP pair.<br>On call to procl<br>stack ← IP<br>IP ← address offset of procl<br>on call to proc2<br>Stack [top] ← CS<br>Stack [top] ← IP<br>CS ← code segment of proc2<br>IP ← address offset of proc2<br>Here we assume that procl is defined within the same segment as the calling procedure, while proc2 is defined in another segment. As far as the calling program is concerned, both the procedures have been called in the same manner. But while declaring these procedures, we declare procl as NEAR procedure and proc2 as FAR procedure, as follows:<br>procl PROC NEAR<br>proc2 PROC FAR<br>LEA BX, procl<br>; initialize BX with the<br>; offset of the procedure<br>; procl<br>CALL BX<br>; CALL procl indirectly<br>; using BX register |
| RET number | It returns the control from | RET 6 |

| | | |
|---|---|---|
| | procedure to calling program. Every CALL should be a RET instruction. A RET instruction, causes return from NEAR or FAR procedure call. For return from near procedure the values of the instruction pointer is restored from stack. While for far procedure the CS:IP pair get is restored. RET instruction can also be followed by a number. | ; In this case, 8086<br>; increments the stack<br>; pointer by this number<br>; after popping off the IP<br>; (for new) or IP and CS<br>; registers (for far) from<br>; the stack. This cancels<br>; the local parameters, or<br>; temporary parameters<br>; created by the<br>; programmer. RET<br>; instruction does not<br>; affect any flags. |
| JMP Label | Unconditionally go to specified address and get next instruction from the label specified. The label assigns the instruction to which jump has to take place within the program, or it could be a register that has been initialised with the offset value. JMP can be a NEAR JMP or a FAR jump, just like CALL. | JMP CONTINUE<br>; CONTINUE is the label<br>; given to the instruction<br>; where the control needs<br>; to be transferred.<br>JMP BX<br>; initialize BX with the<br>; offset of the instruction,<br>; where the control needs<br>; to be transferred. |
| Conditional Jump | All the conditional jumps follow some conditional statement, or any instruction that affects the flag. | MOV CX, 05<br>MOV BX, 04<br>CMP CX, BX<br>; this instruction will set<br>; various flags like the ZF,<br>; and the CF.<br>JE LABEL1<br>; conditional jump can<br>; now be applied, which<br>; checks for the ZF, and if<br>; it is set implying CX =<br>; BX, it makes<br>; a jump to LABEL1,<br>; otherwise the control<br>; simply falls<br>; through to next<br>; instruction<br>; in the above example as<br>; CX is not equal to BX<br>; the jump will not take<br>; place and the next<br>; instruction to conditional<br>; jump instruction will be<br>; executed. However, if<br>; JNE (Jump if not equal<br>; to) or JA (Jump if<br>above), ; or JAE (Jump<br>above or<br>; equal) jump instructions<br>; if applied instead of JE,<br>; will cause the conditional<br>; jump to occur. |
| | All the conditional jump instructions which are given below are self explanatory. | |
| JA/JNBE | Jump if above / Jump if not below nor equal | |

| | | |
|---|---|---|
| JAE/JNB | Jump if above or equal/ Jump if not below | |
| JB/JNAE | Jump if below/ Jump if not above nor equal | |
| JBE/JNA | Jump if below or equal/ Jump if not above | |
| JC | Jump if carry flag set | |
| JE/JZ | Jump if equal / Jump if zero flag is set | |
| JNC | Jump if not carry | |
| JNE/JNZ | Jump if not equal / Jump if zero flag is not set | |
| 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 even | |
| JNP/JPO | Jump if not parity / Jump if parity odd | |
| JG/JNLE | Jump if greater than / Jump if not less than nor equal | |
| JA/JNL | Jump if above / Jump if not less than | |
| JL/JNGE | Jump if less than / Jump if not greater than nor equal | |
| JLE/JNG | Jump if less than or equal to / Jump if not greater than | |
| JS | Jump if sign flag is set | |
| JNS | Jump if sign flag is not set | |
| LOOP label | This is a looping instruction of assembly. The number of times the looping is required is placed in CX register. Each iteration decrements CX register by one implicitly, and the Zero Flag is checked to check whether to loop again. If the zero flag is not set (CX is zero) greater than the control goes back to the specified label in the instruction, or else the control falls through to the next instruction. The LOOP instruction expects the label destination at offset of –128 to +127 from the loop instruction offset. | ; Let us assume we want to<br>; add 07 to AL register,<br>; three times.<br>MOV CX,03<br>; count of iterations<br>L1: ADD AL,07<br>LOOP L1 ; loop back to L1,<br>; until CX<br>; becomes equal to zero<br>; Loop affects no flags. |
| LOOPE/ LOOPZ label | Loop through a sequence of instructions while zero flag = 1 and CX is not equal to zero. There are two ways to exit out of the loop, firstly, when the count in the CX register becomes equal to zero, or when the quantities that are being compared become unequal. | Let us assume we have an array of 20 bytes. We want to see if all the elements of that array are equal to 0FFh or not. To scan 20 elements of the array, we loop 20 times. And we come out of the loop, when either the count of iterations has become equal to 20, or in other words CX register has |

| | | |
|---|---|---|
| | | decremented to zero, which means all the elements of the array are equal to 0FFh, or an element in the array is found which is not equal to 0FFh. In this case, the CX register may still be greater than zero, when the control comes out. This can be coded as follows: (Please note here that you might not understand everything at this place, that is because you are still not familiar with the various addressing modes. Just concentrate on the LOOPE instruction):<br><br>MOV BX, OFFSET ARRAY<br>; Point BX at the start<br>; of the ARRAY<br>DEC BX ; put number of<br>; array elements in CX<br>MOV CX,10<br>L1: INC BX ; point to<br>; next element in array<br>CMP [BX],0FFh<br>; compare array element<br>; with 0FFh<br>LOOPE L1<br>; When the control comes<br>; out of the loop, it has<br>; either scanned all the<br>; elements and found them<br>; to be all equal to 0FFh, or<br>; it is pointing to the first<br>; non-0FFh, element in the<br>; array. |
| LOOPNE/LOOPNZ label | This instruction causes Loop through a sequence of instructions while zero flag = 0 and CX is not equal to zero. This instruction is just the opposite of the previous instruction in its functionality. | |
| JCXZ label | Jump to specified address if CX =0. This instruction will cause a jump, if the value of CX register is zero. Otherwise it will proceed with the next instruction in sequence. | This instruction is useful when you want to check whether CX is zero even prior to entering into a loop. Please note that LOOP instruction executes the loop at least once before decrementing and checking the value of CX register. Thus, CX=0 will execute the loop once and decrement the CX register, |

| | | |
|---|---|---|
| | | making it 0FFFFh, which is non zero. This will cause FFFFh times execution of loop. To avoid such type of conditions you can proceed as follows:<br>JCXZ SKIP _LOOP<br>; if CX is already 0, skip<br>; loop<br>L1: SUB [BX],07h<br>INC BX<br>LOOP L1<br>; loop until CX=0<br>SKIP_LOOP: ..... |

In addition to these instructions, there are other interrupt handling instructions also, which too transfer the control of the program to some specified location. We will discuss these instructions in later units.

### 1.5.5 String Instructions

These are a very strong set of 8086 instructions as these instructions process strings, in a compact manner, thus, reducing the size of the program by a considerable amount. "String" in assembly is just a sequentially stored bytes or words. A string often consists of ASCII character codes. A subscript B following the instruction indicates that the string of bytes is to be acted upon, while "W" indicates that it is the string of words that is being acted upon.

| MNEMONIC | DESCRIPTION | EXAMPLES |
|---|---|---|
| REP | This is an instruction prefix. It causes repetition of the following instruction till CX becomes zero. REP. It is not an instruction, but it is an instruction prefix that causes the CX register to be decremented. This prefix causes the string instruction to be repeated, until CX becomes equal to zero. | REP MOVSB STR1, STR2<br>The above example copies byte by byte contents. The CX register is initialized to contain the length of source string REP repeats the operation MOVSB that copies the source string byte to destination byte. This operation is repeated until the CX register becomes equal to zero. |
| REPE/REPZ | It repeats the instruction following until CX =0 or ZF is not equal to one. REPE/REPZ may be used with the compare string instruction or the scan string instruction. REPE causes the string instruction to be repeated, till compared bytes or words are equal, and CX is not yet decremented to zero. | |
| REPNE/REPNZ | It repeats instruction following it until CX =0 or ZF is equal to 1. This comparison here is just inverse of REPE except for CX, which is checked to be equal to zero. | |
| MOVS/MOVSB/ MOVSW | It causes moving of byte or word from one string to another. This | Assumes both data and extra segment start at address 1000 |

| | | |
|---|---|---|
| | instruction assumes that:<br>• Source string is in Data segment.<br>• Destination string is in extra data segment<br>• SI stores offset of source string in extra segment<br>• DI stores offset of destination string is in data segment<br>• CX contains the count of operation<br>A single byte transfer requires;<br>• One byte transfer from source string to destination<br>• Increment of SI and DI to next byte<br>• Decrement count register that is CX register | in the memory. Source string starts at offset 20h and the destination string starts at offset 30h. Length of the source string is 10 bytes. To copy the source string to the destination string, proceed as follows:<br>MOV AX,1000h<br>MOV DS,AX<br>; initialize data segment and<br>MOV ES,AX<br>; extra segment<br>MOV SI,20h<br>MOV DI,30h<br>; load offset of start of<br>; source string to SI<br>; Load offset of start of<br>; destination string to DI<br>MOV CX,10<br>; load length of string to CX<br>; as counter<br>REP MOVSB<br>; Decrement CX and<br>; MOVSB until<br>; CX =0<br>; after move SI will be one<br>; greater than offset of last<br>; byte in source string, DI<br>; will be one greater than<br>; offset of last destination<br>; string. CX will be equal<br>; to zero. |
| CMPS/CMPSB/ CMPSW | It compares two string bytes or words. The source string and the destination strings should be present in data segment and the extra segment respectively. SI and DI are used as in the previous instruction. CX is used if more than one bytes or words are to be compared, however for such a case appropriate repeating prefix like REP, PEPE etc. need to be used. | MOV CX,10<br>MOV SI,OFFSET SRC_STR<br>; offset of source<br>; string in SI<br>MOV DI, OFFSET DES_STR<br>; offset of destination<br>; string in DI<br>REPE CMPSB<br>; Repeat the comparison of<br>; string bytes until<br>; end of string or until<br>; compared bytes are not<br>; equal. |
| SCAS/SCASB/ SCASW | It scans a string. Compare a string byte with byte in AL or a string word with a word in AX. The instruction does not change the operands in AL (AX) or the operand in the string. The string to be scanned must be present in the extra segment, and the offset of the string must be contained in the DI register. You can use CX if operation is to be repeated using REP prefixes. | MOV AL, 0Dh<br>; Byte to be scanned<br>; for in AL<br>MOV DI,OFFSET DES_STR<br>MOV CX,10<br>REPNE SCAS DES_STR<br>; Compare byte inDES_STR<br>; with byte in AL register<br>; Scanning is repeated while ;<br>the bytes are not equal and ;<br>it is not end of string. If a<br>; carriage return 0Dh is<br>; found, ZF = DI will point ; |

| | | |
|---|---|---|
| | | at the next byte after the ; carriage return. If a ; carriage return is not ; found then, ZF = 0 and ; CX = 0. SCASB or ; SCASW can be used to ; explicitly state whether ; the byte comparison or the ; word comparison is ; required. |
| LODS/LODSB/ LODSW | It loads string byte into AL or a string word into AX. The string byte is assumed to be pointed to by SI register. After the load, the SI pointer is automatically adjusted to point to the next byte or word as the case may be. This instruction does not affect any flag. | MOV SI,OFFSET SRC_STR LODS SRC_STR ; LODSB or LODSW can ; be used to indicate to the ; assembler, explicitly, ; whether it is the byte that ; is required to be loaded or ; the word. |
| STOS/STOSB/ STOSW | It stores byte from AL or word from AX into the string present in the extra segment with offset given by DI. After the copy, DI is automatically adjusted to point to the next byte or word as per the instruction. No flags are affected. | MOVDI,OFFSET DES_STR STOSB DES_STR |

### 1.5.6 Processor Control Instructions

The objectives of these instructions are to control the processor. This raises two questions:

How can you control processor, as this is the job of control unit?
How much control of processor is actually allowed?

Well, 8086 only allows you to control certain control flags that causes the processing in a certain direction, processor synchronization if more than one processors are attached through LOCK instruction for buses etc.

**Note:** Please note that these instructions may not be very clear to you right now. Thus, some of these instructions have been discussed in more detail in later units. You must refer to further readings for more details on these instructions.

| MNEMONIC | DESCRIPTION | EXAMPLE |
|---|---|---|
| STC | It sets carry flag to 1. | |
| CLC | It clears the carry flag to 0. | |
| CMC | It complements the state of the carry flag from 0 to 1 or 1 to 0 as the case may be. | CMC; Invert the carry flag |
| STD | It sets the direction flag to 1. The string instruction moves either forward (increment SI, DI) or backward (decrement SI, DI) based on this flag value. STD instruction does not affect any other flag. The set direction flag causes strings to move from right to left. | |
| CLD | This is opposite to STD, the string | CLD |

| | | |
|---|---|---|
| | operation occurs in the reverse direction.' | ; Clear the direction flag<br>; so that the string pointers<br>; auto-increment.<br>MOV AX,1000h<br>MOV DS, AX<br>; Initialize data segment<br>; and extra segment<br>MOV ES, AX<br>MOV SI, 20h<br>; Load offset of start of<br>; source string to SI<br>MOV DI,30h<br>; Load offset of start of<br>; destination string to DI<br>MOV CX,10<br>; Load length of string to<br>; CX as counter<br>REP MOVSB<br>; Decrement CX and<br>; increment<br>; SI and DI to point to next<br>; byte, then MOVSB until<br>; CX = 0 |

There are many process control instructions other than these; you may please refer to further reading for such instructions. These instructions include instructions for setting and closing interrupt flag, halting the computer, LOCK (locking the bus), NOP etc.

## 1.6 ADDRESSING MODES

The basic set of operands in 8086 may reside in register, memory and immediate operand. How can these operands be accessed through various addressing modes? The answer to the question above is given in the following sub-section. Large number of addressing modes help in addressing complex data structures with ease. Some specific Terms and registers roles for addressing:

Base register (BX, BP): These registers are used for pointing to base of an array, stack etc.
Index register (SI, DI): These registers are used as index registers in data and/or extra segments.
Displacement: It represents offset from the segment address.

Addressing modes of 8086

| Mode | Description | Example |
|---|---|---|
| Direct | Effective address is the displacement of memory variable. | |
| Register Indirect | Effective address is the contents of a register. | [BX] |
| | | [SI] |
| | | [DI] |
| | | [BP] |
| Based | Effective address is the sum of a base register and a displacement. | LIST[BX]<br>(OFFSET LIST + BX) |
| | | [BP + 1] |
| Indexed | Effective address is the sum of an index register and a displacement. | LIST[SI] |
| | | [LIST +DI] |
| | | [DI + 2] |
| Based Indexed | | [BX + SI] |

| | | |
|---|---|---|
| | Effective address is the sum of a base and an index register. | [BX][DI]<br>[BP + DI] |
| Based Indexed with displacement | Effective address is the sum of a base register, an index register, and a displacement. | [BX + SI + 2] |

### 1.6.1 Register Addressing Mode

Operand can be a 16-bit register:

| Addressing Mode | Description | Example |
|---|---|---|
| AX, BX, CX, DX, SI, DI,BP,IP,CS,DS,ES,SS<br>Or it may be AH, AL, BH, BL, CH, CL, DH, DL | In general, the register addressing mode is the most efficient because registers are within the CPU and do not require memory access. | MOV AL,CH<br>MOV AX,CX |

### 1.6.2 Immediate Addressing Mode

An immediate operand can be a constant expression, such as a number, a character, or an arithmetic expression. The only constraint is that the assembler must be able to determine the value of an immediate operand at assembly time. The value is directly inserted into the machine instruction.

MOV AL,05

| Mode | Description | Example |
|---|---|---|
| Immediate | Please note in the last examples the expression (2 + 3)/5, is evaluated at assembly time. | MOV AL,10<br>MOV AL,'A'<br>MOV AX,'AB'<br>MOV AX, 64000<br>MOV AL, (2 + 3)/5 |

### 1.6.3 Direct Addressing Mode

A direct operand refers to the contents of memory at an address implied by the name of the variable.

| Mode | Description | Example |
|---|---|---|
| DIRECT | The direct operands are also called as relocatable operands as they represent the offset of a label from the beginning of a segment. On reloading a program even in a different segment will not cause change in the offset that is why we call them relocatable. Please note that a variable is considered in Data segment (DS) and code label in code segment (SS) by default. Thus, in the example, COUNT, by | MOV COUNT, CL<br>; move CL to COUNT (a<br>; byte variable)<br>MOV AL,COUNT<br>; move COUNT to AL<br>JMP LABEL1<br>; jump to LABEL1<br>MOV AX,DS:5<br>; segment register and<br>; offset<br>MOV BX,CSEG:2Ch<br>; segment name and offset<br>MOV AX,ES:COUNT<br>; segment register and<br>; variable. |

| | | |
|---|---|---|
| | default will be assumed to be in data segment, while LABEL 1, will be assumed to be in code segment. If we specify, as a direct operand then the address is non-relocatable. Please note the value of segment register will be known only at the run time. | ; The offsets of these<br>; variables are calculated<br>; with respect to the<br>; segment name (register)<br>; specified in the<br>; instruction. |

### 1.6.4 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/ arrays etc. For this mode two types of registers are used. These are:

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

BX contain offset/ pointer in Data Segment
BP contains offset/ pointer in Stack segment.
SI contains offset/pointer in Data segment.
DI contains offset /pointer in extra data segment.

There are five different types of indirect addressing modes:

1. Register indirect
2. Based indirect
3. Indexed indirect
4. Based indexed
5. Based indexed with displacement.

| Mode | Description | Example |
|---|---|---|
| Register indirect | Indirect operands are particularly powerful when processing list of arrays, because a base or an index register may be modified at runtime. | MOV BX, OFFSET ARRAY<br>; point to start of array<br>MOV AL,[BX]<br>; get first element<br>INC BX<br>; point to next<br>MOV DL,[BX]<br>; get second element<br>The brackets around BX signify that we are referring to the contents of memory location, using the address stored in BX.<br>In the following example, three bytes in an array are added together:<br>MOV SI,OFFSET ARRAY<br>; address of first byte<br>MOV AL,[SI]<br>; move the first byte to AL<br>INC SI<br>; point to next byte<br>ADD AL,[SI]<br>; add second byte<br>INC SI<br>; point to the third byte<br>ADD AL,[SI]<br>; add the third byte |

| | | |
|---|---|---|
| Based Indirect and Indexed Indirect | Based and indirect addressing modes are used in the same manner. The contents of a register are added to a displacement to generate an effective address. The register must be one of the following: SI, DI, BX or BP. If the registers used for displacement are base registers, BX or BP, it is said to be base addressing or else it is called indexed addressing. A displacement is either a number or a label whose offset is known at assembly time. The notation may take several equivalent forms. If BX, SI or DI is used, the effective address is usually an offset from the DS register; BP on the other hand, usually contains an offset from the SS register. | ; Register added to an offset<br>MOV DX, ARRAY[BX]<br>MOV DX,[DI + ARRAY]<br>MOV DX,[ARRAY + SI]<br>; Register added to a constant<br>MOV AX,[BP + 2]<br>MOV DL,[DI – 2] ; DI + (-2)<br>MOV DX,2[SI] |

| Mode | Description | Example |
|---|---|---|
| Based Indexed | In this type of ad lressing the operand's effective address is formed by combining a base register with an index register. | MOV AL,[BP] [SI]<br>MOV DX,[BX + SI]<br>ADD CX,[DI] [BX]<br>; Two base registers or two<br>; index registers cannot be<br>; combined, so the<br>; following would be<br>; incorrect:<br>MOV DL,[BP + BX]<br>; error : two base registers<br>MOV AX,[SI + DI]<br>; error : two index registers |
| Based Indexed with Displacement | The operand's effective address is formed by combining a base register, an index register, and a displacement. | MOV DX,ARRAY[BX][SI]<br>MOV AX, [BX + SI + ARRAY]<br>ADD DL,[BX + SI + 3]<br>SUB CX, ARRAY[BP + SI]<br>Two base registers or two index registers cannot be combined, so the following would be incorrect:<br>MOV AX,[BP + BX + 2]<br>MOV DX,ARRAY[SI + DI] |

## ☞ Check Your Progress 3

State True or False. | T | F |

1. CALL instruction should be followed by a RET instruction. ☐

2. Conditional jump instructions require one of the flags to be tested. ☐

3. REP is an instruction prefix that causes execution of an instruction until CX value become 0. ☐

4. In the instruction MOV BX, DX register addressing mode has been used. ☐

5. In the instruction MOV BX,ES:COUNTER the second operand is a direct operand. ☐

6. In the instruction ADD CX, [DI] [BX] the second operand is a based index operand, whose effective address is obtained by adding the contents of DI and BX registers. ☐

7. The instruction ADD AX,ARRAY [BP + SI] is incorrect. ☐

## 1.7 SUMMARY

In this unit, we have studied one of the most popular series of microprocessors, viz., Intel 8086. It serves as a base to all its successors, 8088, 80186, 80286, 80486, and Pentium. The successors of 8086 can be directly run on any successors. Therefore, though, 8086 has become obsolete from the market point of view, it is still needed to understand advanced microprocessors.

To summarize the features of 8086, we can say 8086 has:

- a 16-bit data bus
- a 20-bit address bus
- CPU is divided into Bus Interface Unit and Execution Unit
- 6-byte instruction prefetch queue
- segmented memory
- 4 general purpose registers (each of 16 bits)
- instruction pointer and a stack pointer
- set of index registers
- powerful instruction set
- powerful addressing modes
- designed for multiprocessor environment
- available in versions of 5Mhz and 8Mhz clock speed.

You can refer to further readings for obtaining more details on INTEL and Motorola series of microprocessors.

## 1.8 SOLUTIONS/ANSWERS

### Check Your Progress 1

1. It improves execution efficiency by storing the next instruction in the register queue.

2. a) 0100 × 10h ( - 16 in decimal) + 0020h
   = 01000h + 0020h
   = 01020h

   b) 0200h × 10h + 0100h
   = 02000h + 0100h
   = 02100h

   c) 4200h × 10h + 0123

= 42000h + 0123h
= 42123h

3. a) False b) True c) True d) False

## Check Your Progress 2

1. (a) PUSHF instructions do not take any operand.
   (b) No error.
   (c) XCHG instruction cannot have two memory operands
   (d) AAA instruction performs ASCII adjust after addition. It is used after an ASCII Add. It does not have any operands.
   (e) IDIV assumes one operand in AX so only second operand is needed to be specified.

2. (a) False
   (b) False
   (c) True
   (d) False
   (e) False

## Check Your Progress 3

1. False
2. True
3. True
4. True
5. True
6. True
7. False