Unit - 3

A) Data Movement instructions of 8086 :-

The instruction of the 8086 are classified as data transfer, arithmetic, logical, flag manipulation, control transfer, shift / rotate, string and machine control instructions.

The data transfer instructions include MOV, PUSH, POP, XCHG, IN, OUT, LEA, LDS, LSS, LAHF & SAHF

i) MOV :- The MOV instruction copies a word or byte of data from a specified source to a specified destination. The destination can be a register or a memory location. The source can be a register, a memory location, or an immediate number.

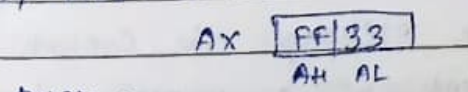
The general format of the MOV instruction is MOV destination, source.

Eg :-

a) MOV BL, 50H : Move immediate data 50H to BL

b) MOV CX, [BX] : Copy the word from the memory at [BX] to CX.

c) MOV AX, CX : Copy the contents of CX to AX

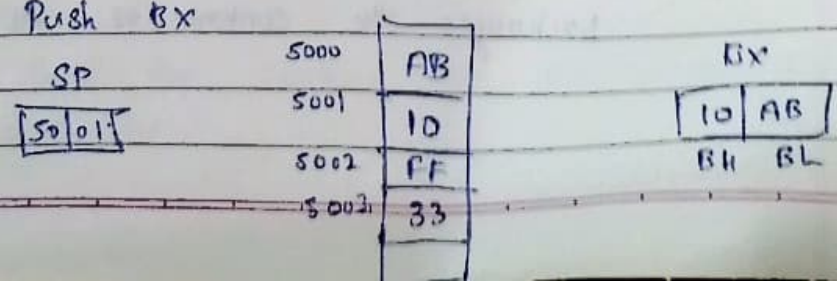


ii) PUSH :-

The Push instruction is used to store the word in a register or a memory location into the stack, as explained in the stack addressing mode. SP is decremented by two after the execution of PUSH.

a) Push cx :- Push the content of CX into the stack.

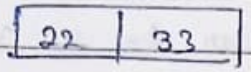
Eg :- Push BX

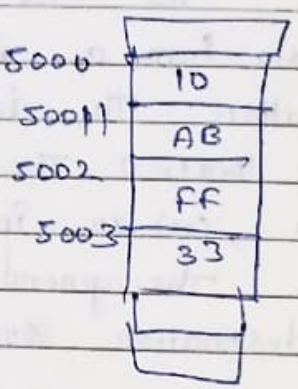


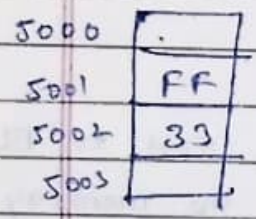
iii) Pop :- The Pop instruction copies the top word from the stack to a destination specified in the instruction. The destination can be a general purpose register, a segment register, (or) a memory location. After the word is copied to the specified destination, sp is incremented by two.

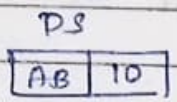
a) Pop BX :- Pop the content of BX from the stack

Pop DS

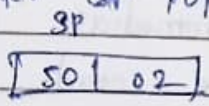








After Pop the sp will be decremented



ii) XCHG :-

The XCHG instruction exchanges the contents of a register with the contents of a memory location. It cannot exchange the contents of two memory locations directly. The source and destination must both be either word (or) bytes.

The segment register cannot be used in the instruction.

a) XCHG AL, BL

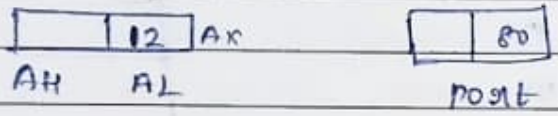
Exchanges the content of AL & BL.

5) IN :-

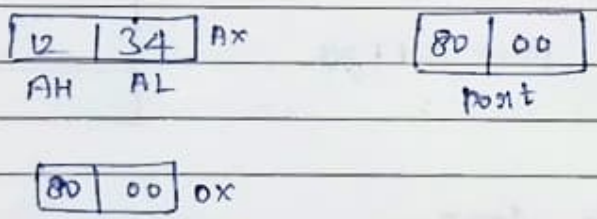
The IN instruction copies data from a port to the AL or AX register. If a 8-bit port is read, the data is stored in AL and if a 16-bit is read, the data is stored in AX.

The IN instruction has two formats fixed port & Variable port.

IN AL, 80H : Input a byte from the port with address 80H to AL.



IN Ax, DX



6) OUT :-

The OUT instruction transfers a byte from AL or a word from AX to the specified port. Similar to IN instruction, the OUT instruction has two forms. fixed port & Variable port

OUT 48H, AL

Sends the content of AL to the port with address 48H.

OUT 0F0H, AX

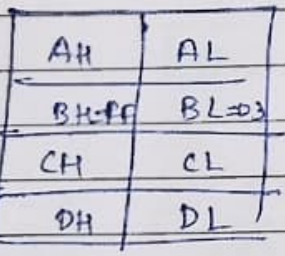
Send the content of Ax to the port with address F0H

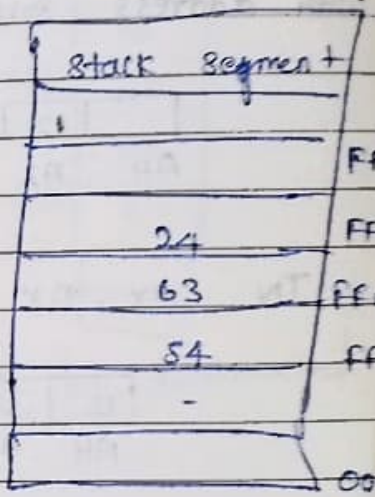
7) LEA (load effective address)

The general format of the LEA instruction is LEA register, source. This instruction determines the offset address of the variable (or) memory location called the source & puts this offset address in the indicated 16-bit register.

Eg:- Price is a displacement

LEA Bx, price





Displacement price = FF03

LEA Bx, COST

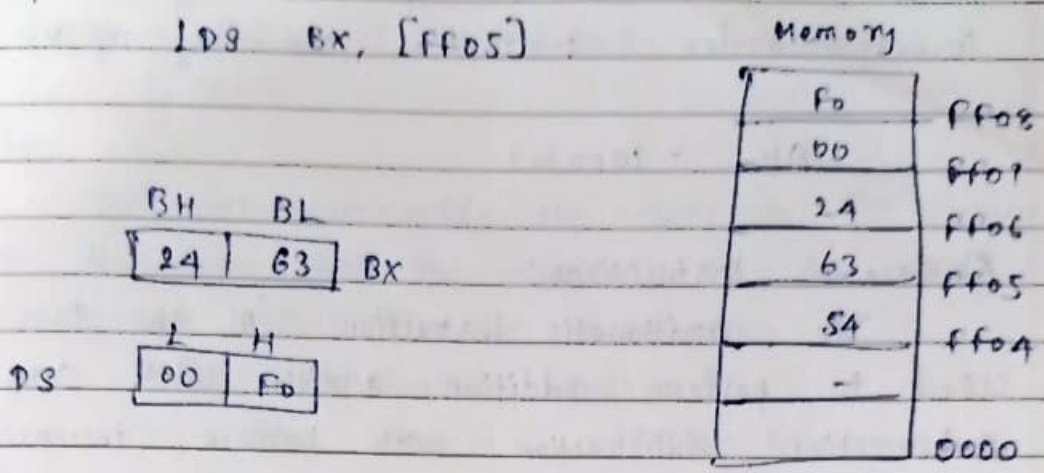
Load Bx with the offset address of COST in the data segment, where COST is the name assigned to a memory location in the DS.

8) LDS :-

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

Example :-

LDS BX, [FF05]

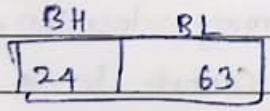


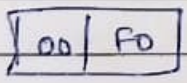
9) LES :-

LES & LSS instruction are similar to the LDS instruction, except that instead of the DS register, the ES & SS registers respectively, are along with the register specified in the instruction.

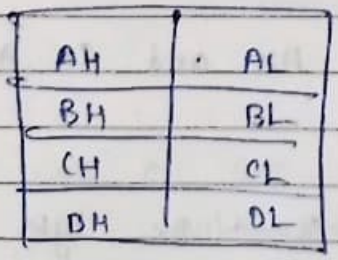
eg:- LES DI, [BX]

LES BX, [FF05]





10) LAHF :- This instruction copies the lower-order byte of the flag register in to AH.



Flag register

HB

O - Overflow

D - Direction

I - Interrupt

T - Trap

LB

S - Sign

Z - Zero

Ac - Auxilary Carry

P - Parity

C - Carry

ii) SAHF :- Store contents of AH register into lower order 8-bits of the flag register.

AH = 10001011

*) Arithmetic Instructions :-

The arithmetic instructions in the 8086 are used to perform addition, addition with carry, substraction, substraction with borrow, increment, decrement, comparison, multiplication, division, & processing of Ascii data.

1) i) ADD :-

The general format of the ADD instruction is ADD destination, source.

The data from the source & destination are added and the result is placed in the destination.

The source may be an immediate number, a register, a memory location. The destination can be a register (or) a memory location. However the source & destination cannot be a memory location.

eg :- i) ADD

a) ADD BL, 80H

Add the immediate data 80H to BL

b) ADD CX, 12B0H

Add the immediate data 12B0H to CX

c) ADD AX, CX

Add the content of AX and CX & store the result in AX.

d) ADD AL, [BX]

Add the content of AL & the byte from the memory at [BX] & store the result in AL

The flags AF, CF, OF, PF, SF & ZF are affected by the execution of the ADD instruction.

ii) ADC :-

This instruction adds the data in the source and destination with the content of the carry flag and stores result in the destination.

ADC destination, source

ADC 0100H , ADC AX, BX.

2) Substraction :-

The general form of the substract (SUB) instruction is SUB destination, source. It substracts the number in the source from the number in the destination and stores the result in the destination.

Like the ADD instruction, the source may be an immediate number, a register, or a memory location. The destination can be a register (or) a memory location. However source & destination cannot be a memory location.

For substraction, the carry flag (CF) functions as the borrow flag. If the result is negative after substraction, CF is set, otherwise it is reset. The flags AF, CF, OF, PF, SF & ZF are affected by SUB instruction.

eg:- i) SUB

a) SUB AL, BL

Substract BL from AL & store the result in AL.

b) SUB CX, BX

Substract BX from CX & storre the result in CX.

c) SUB Bx, [DI] :-
Subtract the word in the memory at [DI] from Bx & store the result in Bx.

ii) SBB :- Subtract with borrow.
The SBB instruction substracts the content of the source and the carry flag from the content of the destination & stores the result in the destination.
AF, CF, OF, PF, SF & ZF are affected by the instruction.

SBB AX, BX

d) Multiplication :-
i) MUL :-
The multiply (MUL) instruction is used for multiply two unsigned bytes (or) words. The general form of the MUL instruction is MUL source. The source can be a byte (or) a word from a register or memory location. which is conside as the multiplier.
The multiplicand is taken by default from AL & AX. The result of multiplication is stored in AX & DX/AX

eg:- MUL → unsigned byte (0 to 255).
Two rules.
Multiplication is done with AL & AX
AL is 8-bit called byte
Ax is 16 bit called word

4-Nibble
8-byte
16-word
32-double word

1) AL multiply any number with AL output stored in AX

2) Multiply number with AX, the output stored in DX/AX. DX store higher word & AX store lower word

MUL BL :- AL x BL, output store in AX
MUL CH :- AL x CH, output store in AX
MUL BX :- AX * BX, output store in DX & AX

ii) IMUL :- (-128 to 127)

This instruction is used for multiplying the signed byte (or) word in a register (or) memory location with AL (or) AX. and store the result in Ax (or) DX/Ax respectively.

eg:-

```
MOV  AL, -2
MOV  BL, -4
IMUL BL        AX = 8
```

To multiply a signed byte by a signed word, the byte is moved into a word location and the upper byte of the word is filled with copies of the sign bit. If the byte is moved into AL, using the CBW [convert byte to word] instruction, the sign bit in AL is extended into all the bits of AH. Thus, Ax contains the 16-bit sign extended word.

*) Division :-

i) DIV :- The divide (DIV) instruction is used for dividing unsigned data. The general form of the DIV instruction is DIV source.
When source is the divisor. It can be a byte (or) word in a register (or) memory location. The dividend is taken by default from Ax and Dx/Ax for byte (or) word type data division.

Rules :-

1) When a word divide by a byte
16-bit 8-bit
word will be taken from Ax
AL will store 8-bit Quotient
AH will store 8-bit remainder

Dividend (bits)	Divisor	Quotient	Remainder
AX (16)	Source (8)	AL (8)	AH (8)

eg :- Mov Ax, 203
mov Bl, 4
DIV BL

Ax / BL
= 203/4

Quotient AL = 50
Remainder AH = 03

2) When a 32-bit is divided by 16-bit :
The MSB of 32-bit stored in - DX
The LSB of 32-bit stored in AX
Ax will store 16-bit Quotient
Dx will store 16-bit Remainder

Dividend (bits)	Divisor (bits)	Quotient (bits)	Remainder (bits)
DX-AX (32)	Source (16)	AX (16)	DX (16)

ii) IDIV :-

The IDIV instruction is used for dividing signed data. The general form and the rules for IDIV instruction are same as those for the DIV instruction. The Quotient is a signed number and the sign of the remainder is the same as the sign of the dividend.

eg:- MOV AX, -203 Quotient AL = -50
MOV BL, 4 Remainder AH = 03
IDIV BL

AL will store 8-bit Quotient
AH will store 8-bit remainder

iii) Compare :-

The general form of the compare instruction is CMP destination, source. This instruction compares a byte (or) word in the source with a byte (or) word in the destination and affects only the flags, according to the result. The content of the source & destination are not affected by the execution of this instruction. The comparison is done by subtracting the content of the source from the destination.

The rules for the source & destination are the same as those for the SUB instruction.

Case : 1 CMP DL, CH.
if DL > CH.
eg: 5-2 = 3

CF	ZF	SF
0	0	0

FF = 1111 1111
E7 1110 0111
0001 1000

case - ii)

If DL = CH

eg :- 5-5 = 0

CF	ZF	SF
0	1	0

E7 = 1110 0111
E7 = 1110 0111
0000 0000

case - iii)

If DL < CH

eg :- 3-5 = -2

CF	ZF	SF
1	0	1

*) BCD (Binary Coded Decimal).

i) DAA :-

Decimal adjust after BCD addition. This instruction is used to get the result of addition of two packed BCD numbers

The result of addition must be in AL for DAA to work correctly.

If the lower nibble in AL is greater than 9 after addition (or) if the AF flag is set by the addition, the DAA instruction adds 6 to the lower nibble in AL.

If the result in the upper nibble of

AL is now greater than 9 (or) if the Carry flag is set by the addition, the DAA instruction adds 60H to AL.

eg :- (1)

Let AL = 58 BCD
CL = 35 BCD

```
  0101 1000
  0011 0101
  ---------
  1000 1101
```

Consider the execution of the following instructions

ADD AL, CL

after addition AL = 8DH and AF = 0

DAA instruction adds 0110 (decimal 6) to AL, Since lower nibble in AL is greater than 9.

```
8D  = 1000 1101
      0000 0110
      ---------
      1001 0011   = 93 BCD and CF = 0.
```

```
MOV AL, 58
MOV CL, 35
ADD AL, CL
DAA
HLT.
```

eg :- (2)

Let AL = 88 BCD
CL = 49 BCD

```
  1000 1000      AF = set to 1
  0100 1001
  ---------
  1101 0001
```

After using instruction DAA. The upper nibble is greater than 9. So add 0110 (decimal 6) to upper nibble

0001
(1) 0110 0000
0011 0001

Eg :2

ii) DAS :- Decimal adjust after BCD Subtraction

The result of the subtraction must be in AL for DAS to work correctly.

If the lower nibble in AL after a subtraction is greater than 9 (or) if the AF is set by subtraction, the DAS instruction subtracts 6 from the lower nibble of AL.

If the result in the upper nibble is now greater than 9 (or) if the Carry flag is set, the DAS instruction subtracts 60H from AL.

eg :-

Let AL = 86
CH = 57

HN LN
1000 0110
0101 0111
0001 1111
2 F

Lower nibble is greater than 9 Substract 06 from AL to make AL = 00101001 = 29.
The result is 29 = AL.

```
MOV AL, 86
MOV CH, 57
SUB AL, CH
DAS
HLT:
```

Ascii instructions :-

i) AAA :- Ascii adjust after Addition

The instruction must always follow the addition of two unpacked BCD operands in AL.

When AAA is executed, the content of AL is changed to a valid unpacked BCD number; the upper four bits of AL are cleared. CF is set and AH is incremented if a decimal carry-out from AL is generated.

eg :-
```
MOV AL, 80          80 = 1000 0000
ADD AL, C5          C5 = 1100 0101
AAA                 (1) 0100 0101   C=1
HLT                      4    5     A=1
```

If the addition of two numbers produces a decimal carry, the AH register incremented by 1, and the CF and AF flags are set.

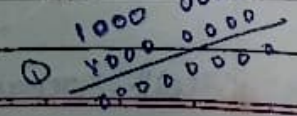
If there was no decimal carry, the CF and AF flags are cleared and the AH register is unchanged.

In either case, bits 4 through 7 of the AL registers are cleared to 0.

In above example, the result will be AL = 0000 0101 = 05 & AH = 01.

eg : 2
```
MOV AL, 80          80 = 1000 0000
ADD AL, 45          45 = 0100 0101
AAA                 1100 0101   C=0
HLT.                 C    5     A=0
                     0    5
```

Here the result will be AL = 0000 0101 = 05 & AH = No change. flag registers no change.



ii) AAS :- ASCII adjust after Subtraction :-

This instruction always follows the subtraction of one unpacked BCD operand from another in AL.

It changes the content of AL to a valid unpacked BCD number and clears the top four bit of AL. CF is set and AH is decremented if a decimal borrow occurs.

Eg :-

```
MOV AL, 88          88 = 1000   1000
SUB AL, C1          C1 = 1100   0001
AAS                 1100        0111
HLT                 C           7
```

Borrow = 1

CF = 1
AF = 1

Data in AL is 88 and AH = 4

If the subtraction produces a decimal carry, the AH register is decremented by 1 and the CF and AF flags are set.

If there was no decimal carry the CF and AF flags are cleared and the AH register is unchanged.

In either case, bits 4 through 7 of the AL register are cleared to 0.

In above example, the result will be AL = 07 and AH = 3.

Another eg without borrow :-

```
MOV AL, 88          88 = 1000   1000
SUB AL, 08          08 = 0000   1000
AAS                 Carry = 0.  1000   0000
HLT                             8      0
```

AF = 0
CF = 0

The result will be AL = 00 and AH will remain same, no change.

*) 8086 Assembler Directives:-

Assembly languages are low level languages for programming computers, microprocessors, microcontrollers and other IC.

An assembler directive is a statement to give direction to the assembler to perform task of the assembly process.

It controls the organization is the program and provide necessary information to the assembler to understand the assembly language programs to generate necessary machine codes. They indicate how an operand (or) a section of the program is to be processed by an assembler.

An assembler supports directives to define data, to organize segments to control procedure, to define macros. The directives are not translated in to machine codes.

They do not have operation codes.

Assembler Directive for variable & constant definition

Assembler directive helps the assembler to correctly understand the assembly language programs to prepare the codes.

1) DB directive :- Define byte.

The DB directive is used to declare a BYTE - 2BYTE variable.

A BYTE is made up of 8-bits.

Eg :-

DATA1 DB 20H.

Reserve one byte for storing DATA1 and assign value 20H to it.

2) DW directive :- Define word

DW directive is used to declare a WORD type variable

A word occupies 16 bits or (2 BYTE)

Eg:-

DATA DW 1020H - Reserve one word for storing DATA 2 and assign the value 1020 to it.

3) DD:-

The DD directive is used to declare a Double words

It is made up of 32 bits.

2 word's (or) 4 byte.

eg:-

DATA 3 DD 1234ABCDH.

Intialize Data 3 as a double word with 1234ABCD H

4) DQ:- Define Quad word = 4 word

Eg:-

DATA 4 DQ 1234ABCD5678EFBBH.

Initialize DATA 4 as a quad word with 1234ABCD5678EFBBH.

5) EQU:-

The directive EQU is used to assign a value to a data name.

eg:-

NUMBER EQU 50H

Assign the value 50H to number.

Assembler Directives Related to Code location:-

i) ORG:-

The ORG directive directs the assembler to start the memory allocation for a particular segment from the declared address in the ORG statement.

eg:- ORG 100H

When this directive is placed at the beginning, the location counter is initialized with 0100H.

2) Segment and ends:-

The segment and END directive indicate the start and end of a segment.

Segment may be assigned a type such as PUBLIC (it can be used by any other modules of the program while linking)

GLOBAL (it can accessed by any other module)

Eg:-

```
CODE1  Segment
:
:          Instructions of CODE1 Segment
:
CODE1  Ends.
```

ASSUME:-

The assume directive is used to inform the assembler, the name of the logical segments to be assumed for different segment used in the program.

eg:-

ASSUME CS: CODE1, DS: DATA1

This statement informs the assembler that the segment address where the logical segment CODE1 and DATA1 are loaded in memory during execution is to be stored in the CS and DS registers respectively.

GROUP:-

This directive is used to form a logical group of segments with a similar purpose.

eg:-

Program1 Group code1, DATA1, STACK1

EXTRN and Public :-

The directive EXTRN informs the assembler that the procedures, labels, and names declared after this directive have already been defined in some other segments and in the segments where they actually appear, They must be declared PUBLIC, using the public directive

eg :-

```
MODULE 1 SEGMENT
Public  Square_root
.
.
                    // code of Square_root
Square root END
MODULE 1 ends

Module 2  Segment
EXTRN  Square_Root
.
.
                    // Code of Module_2.
CALL Squrare_root
Module_2 ENDS
```

The remaining assembler directives refer microprocessor and microcontroller Book.