Chapter Three

Programming the 8086 Microprocessor

3.1 Assembly Language

Assembly language is the programming language of the microprocessor that uses the of abbreviation of the names of the operations that deals different types of data and understood by the human. Each type of CPU has its own assembly language, so this assembly language program was written for one type of CPU and won't run on another.
In assembly language, a unique letters combination is assigned for each operation. These letters combination is referred to as mnemonic such as:

MOV refers to *Move* operation.

ADD refers to *Addition* operation.

MUL refers to *Multiplication* operation.

Assembly language programs are translated into machine language by a program called an assembler.

3.2 Instruction Set

There are a total of 117 basic instructions for the 8086. The wide range of operands and addressing modes permitted for use with these instructions further expands the instruction set into many more executable instruction. For e.g., the basic MOV instruction expands into 28 different machine level instructions.

The instruction set will be divided into a number of groups of functionally related instruction, these are:

- Data transfer instructions.
- Arithmetic instructions.
- Logical instructions.
- Shifting and Rotating instructions.

- Flag control instructions.
- Program flow control instructions.
- String instructions.
- Miscellaneous instructions.

3.2.1 Data transfer instructions

The 8086 MP has a group of data transfer instructions that are provided to move data either between its internal registers or between an internal register with a storage memory location. In this section we will introduce some of the instructions used in data transfer. These are (**MOV, XCHG, LEA, LDS, LES**). The other instructions will be studied in the future.

i. MOV instruction

The MOV instruction is used to transfer 8 and 16-bit data to and from registers. Either the *source* or *destination* has to be a register. The other operand can come from another ***register***, from ***memory***, from ***immediate data*** (a value included in the instruction).

Mnemonics	Meaning	Format	Operation	Flags affected
MOV	Move	MOV Dest.,Source	(Source) ⟶ (Dest.)	None

The large choice of source and destination results in many different move instructions. Table (3.1) shows the valid source and destination variations.

Destination	Source
Mem.	Accumulator
Accumulator	Mem.
Reg.	Reg.
Reg.	Mem.
Mem.	Reg.
Reg.	Immediate
Mem.	Immediate
Seg. – Reg.	Reg. – 16
Seg. – Reg.	Mem. – 16
Reg. – 16	Seg. – Reg.
Mem.	Seg. – Reg.

Table (3.1) ***Valid Source and Destination Variations used in MOV Instruction***

Examples:

MOV AL, 7Eh ; immediate mode.

MOV CX,DX ; Register mode.

MOV [1000h], CX ; Direct mode.

MOV [SI],AX ; Register Indirect mode.

MOV DI, [BP+SI+2F3Eh] ; Base Relative Plus Index.

ii. XCHG instruction

The 8086 XCHG instruction ***swaps*** the content of the source with content of the destination. The data can be swapped either between 2 general purpose registers or between a general purpose registers and a storage memory location.

Mnemonics	Meaning	Format	Operation	Flags affected
XCHG	Exchange	XCHG Dest.,Source	(Source) ⟷ (Dest.)	None

Table (3.2) shows the types of operands that can be used with XCHG instruction.

Destination	Source
Mem.	Accumulator
Accumulator	Mem.
Reg.	Reg.
Reg.	Mem.
Mem.	Reg.

Table (3.2) ***Valid Source and Destination Variations used in XCHG Instruction***

Example:

XCHG CX,DX ; Register mode.

XCHG [1000h], CX ; Direct mode.

XCHG [SI],AX ; Register Indirect mode.

XCHG DI, [BP+SI+2F3Eh] ; Base Relative Plus Index.

Important note: *immediate addressing mode* is not allowed in **XCHG** instruction.

iii. LEA instruction

The LEA instruction is used to load a 16-bit Reg. with a 16-bit offset of a memory location.

Mnemonics	Meaning	Format	Operation	Flags affected
LEA	Load Effective Addres	LEA Reg 16, [offset]	offset ⟶ Reg 16	None

The destination operand can be one of the 16-bit general purpose registers while the source operand is an offset address that can be used by any addressing mode.

Examples:

LEA DI, [2F3Eh] ; Direct.

LEA CX, [BP+SI+2F3Eh] ; Base Relative Plus Index.

LEA SP, [BX] ; Register indirect.

LEA DX, [BX+2000h] ; Register Relative.

Important note: *immediate addressing mode* is not allowed in **LEA** instruction.

iv. LDS and **LES** instructions

The LDS instruction loads any 16-bit register with an offset address and the **DS** register with segment address. For example:

LDS CX, [DI]

this instruction transfers the 32-bit addressed by DI in the data segment into the CX and DS registers respectively.

The LES instruction is similar to the LDS instruction except it loads the Extra segment register ES instead of DS.

Example 1: assume DS=2700h, DI=5000h, and SI=C500h, what are the new values of DS and SI after executing the following instruction

LDS SI, [DI+02h]

Assume the content of memory locations are specified as shown below:

The PA of the memory location specified by the instruction is:

PA=DS*10h+(DI+02h)
= 27000+5000+02
=2C002h

So the registers content after the execution of the instruction will be as follows:
SI= BE3Ch
DS= 5020h

[2C002]	3C
[2C003]	BE
[2C004]	20
[2C005]	50

3.2.2 Arithmetic Instructions

The 8086 MP has a group of arithmetic instructions that perform different types of arithmetic operations. The arithmetic operations such as Addition (**ADD**, **ADC**, and **INC**) instructions, Subtraction (**SUB**, **SBB**, **DEC**, and **NEG**) instructions, Multiplication (**MUL** and **IMUL**) instructions, and Division (**DIV** and **IDIV**) instructions. The status flags (CF, AC, SF, PF, ZF, OF) are affected by the arithmetic operations and changed depending on the results of the operations.

3.2.2.1 Addition instructions

i. ADD, ADC instructions

The **ADD** instruction is used to perform the addition between two operands (i.e. between the Destination and Source) operands. It simply adds the content of the source to the content of the destination and stores the result into the destination operand. The status flags that will be affected by the ADD instruction are CF, AC, SF, PF, ZF, OF.

Examples:

ADD AL, BL ; $AL_{new}=AL_{old}+BL$.

ADD SI, [3000h] ; $SI_{new}=SI_{old}$+the content of the M.L. with offset of 3000h.

ADD CX, 5000h ; $CX_{new}=CX_{old}+5000h$

ADD BYTEPTR[BP+5020h], 3Eh ; the content (8-bit) of M.L. with offset of (BP+5020h)+3Eh.

The **ADC** (ADD with carry) instruction has the same operation of the ADD instruction but will add the content of the **old CF** to the result. The status flags that will be affected by the ADD instruction are CF, AC, SF, PF, ZF, OF.

Examples:

ADC SI, DI ; $SI_{new}=SI_{old}$+DI+old CF

ADC DX, [BX] ; $DX_{new}=DX_{old}$+ the content of the M.L. with offset of BX+old CF

ADC WORDPTR[2000h], 6600h ; the content of M.L. (16-bit) with offset of (2000h)+6600h+ old CF.

Important note: in case of addition with *immediate addressing mode*, it is acceptable to add the (8-bit) number to the (16-bit) contents

Example:

ADD CX, 50h

ADC SI, FFh

ii. INC (increment) instruction is considered as one of the addition arithmetic instructions. It simply adds ***one*** to the content of the register or the content of the memory locations. It takes ***One operand only that represents the Source and Destination*** in the same time. It also affects the status flags AC, SF, PF, ZF, OF except the CF.

Examples:

INC BYTEPTR[BX+SI+5000h] ; the content of the M.L. (8-bit) with offset of (BX+SI+5000h)+1

INC CX ; $CX_{new}=CX_{old}+1$

Important note: the **INC** instruction does not take the *immediate addressing mode*.

Table below shows the operations of the addition instructions.

Mnemonics	Meaning	Format	Operation	Flags affected
ADD	Addition	ADD D,S	D+S ⟶ D	CF, AC, SF, PF, ZF, OF
ADC	ADD with Carry	ADC D,S	D+S+old CF ⟶ D	CF, AC, SF, PF, ZF, OF
INC	ADD 1 to the content	INC D	D+1 ⟶ D	AC, SF, PF, ZF, OF

Note: in the above table D means Destination and S means Source.

Example 2: Write an Assembly Language Program (A.L.P.) to add with carry two consecutive bytes (8-bit) of data stored in data segment of start address of 7000h and an offset specified by [BX+SI]. Store the result in register BL.

Solution:

```
MOV AX, 7000h
MOV DS, AX
MOV BL, 00h
ADC BL, [BX+SI]
ADC BL, [BX+SI+01h]
HLT
```

H.W. 1: Write an Assembly Language Program (A.L.P.) to add two consecutive words (16-bit) of data stored in data segment of start address of 2000h and an offset specified by the PA [27000h]. Store the result in the next 2 M.L.s followed the 2 consecutive words.

3.2.2.2 Subtraction instructions

i. SUB, SBB instructions

The **SUB** instruction is used to perform the subtraction between two operands (i.e. between the Destination and Source) operands. It simply subtracts the content of the source from the content of the destination and stores the result into the destination operand. The status flags that will be affected by the SUB instruction are CF, AC, SF, PF, ZF, OF.

Examples:

SUB AL, BL ; $AL_{new}=AL_{old}$ – BL ($AL_{new}=AL_{old}$ + 2's comp. of BL)

SUB SI, [3000h] ; $SI_{new}=SI_{old}$ – the content of the M.L. with offset of 3000h (SI_{old} +2's comp. of the content of the M.L. with offset of 3000h)

SUB CX, 5000h ; $CX_{new}=CX_{old}$ – 5000h (CX_{old} + the 2's comp. of the number 5000h).

SUB BYTEPTR[BP+5020h], 3Eh ; the content (8-bit) of M.L. with offset of (BP+5020h) – 3Eh(the content (8-bit) of M.L. with offset of (BP+5020h) + 2's comp. of the number 3Eh)

The **SBB** (SUB with Borrow) instruction has the same operation of the SUB instruction but will subtract the content of the **old CF(borrow)** from the result. The status flags that will be affected by the SBB instruction are CF, AC, SF, PF, ZF, OF.

Examples:

SBB SI, DI ; $SI_{new}=SI_{old}$–DI– old CF (SI_{old} + 2's comp. of (DI+old CF))

SBB DX, [BX] ; $DX_{new}=DX_{old}$ – the content of the M.L. with offset of BX– old CF (DX_{old} + the 2's comp. of (the content of the M.L. with offset of BX+ old CF))

SBB WORDPTR[2000h], 6600h ; the content of M.L. (16-bit) with offset of (2000h) – 6600h+ old CF(the content of M.L. (16-bit) with offset of (2000h) + the 2's comp. of (6600h+ old CF))

Important note: in case of subtraction with *immediate addressing mode*, it is acceptable to subtract the (8-bit) number to the (16-bit) contents

Examples:

SUB CX, 50h

SBB SI, FFh

ii. **DEC** (decrement) instruction is considered as one of the subtraction arithmetic instructions. It simply subtracts ***one*** from the content of the register or the content of the memory locations. It takes ***One operand only that represents the Source and Destination*** in the same time. It also affects the status flags AC, SF, PF, ZF, OF except the CF.

Examples:

DEC BYTEPTR[BX+SI+5000h] ; the content of the M.L. (8-bit) with offset of (BX+SI+5000h) – 1

DEC CX ; $CX_{new}=CX_{old}$ – 1

Important note: the **DEC** instruction does not take the *immediate addressing mode*.

iii. **NEG** (negate or negative) instruction ***inverts the sign of the content*** of either the M.L. or of a register. It actually takes the ***2's complement of the content***.

Examples:

NEG AX ; $AX_{new} = -AX_{old}$ (2's comp. of AX).

NEG WORDPTR[BP+DI] ; – the content (16-bit) of the M.L. with offset of (BP+DI) (2's comp. of the content (16-bit) of the M.L. with offset of (BP+DI)).

Important note: the **NEG** instruction does not take the *immediate addressing mode*.

Table below shows the operations of the subtraction instructions.

Mnemonics	Meaning	Format	Operation	Flags affected
SUB	Subtraction	SUB D,S	D – S ⟶ D	CF, AC, SF, PF, ZF, OF
SBB	SUB with Carry	SBB D,S	D – S – old CF ⟶ D	CF, AC, SF, PF, ZF, OF
DEC	SUB 1 from the content	DEC D	D – 1 ⟶ D	AC, SF, PF, ZF, OF
NEG	Negate	NEG D	2's comp. (D)	CF, AC, SF, PF, ZF, OF

Example 3: Trace the following program step by step showing the values of the (CF, ACF, PF, ZF, SF) assuming all initial values of the flags are ZEROS:

```
MOV AX, 1623h
MOV BX, F014h
MOV CX, 2002h
ADD AX, BX
DEC BX
SBB BX,CX
NEG CX
HLT
```

Solution:

STEP	AX	BX	CX	CF	ACF	PF	SF	ZF
1	1623	0000	0000	0	0	0	0	0
2	1623	F014	0000	0	0	0	0	0
3	1623	F014	2002	0	0	0	0	0
4	0637	F014	2002					
5	0637	F013	2002					
6	0637	D010	2002					
7	0637	D010	DFFE					

H.W. 2: Trace the following program step by step showing the values of the (CF, ACF, PF, ZF, SF) assuming all initial values of the flags are ZEROS:

MOV BX, 1234h

MOV DX, 2500h

INC CX

NEG CX

ADC DX, CX

ADD CX,32h

HLT

H.W. 3: Write an A.L.P to perform the following operations:

M1= M2+5

M2=M1–12

M3=M2+25

M4= –M3

Where M1, M2, M3, and M4 are memory locations that have an offsets of 0300h, 0400h, 0500h, and 0600h respectively.

3.2.2.3 Multiplication instructions: MUL and IMUL

The **MUL** (for unsigned numbers) and **IMUL** (for signed numbers) instructions are used to perform the multiplication operation between two operands: the *source* is external and the *destination* is internal. The source may be either register or memory with data either 8-bit or

16-bit. The internal destination could be either **AX** for an 8-bit source operand or the registers pair **DX AX** for a 16-bit source operand. This operation is explained in table below:

Multiplication (MUL or IMUL)	**Internal operand**	**External operand (source)**	**Result**
Byte * Byte	AL	Register or BYTEPTR of memory	AL*source = AX
Word * Word	AX	Register or WORDPTR of memory	AX*source = DX AX

The general format for the multiplication instructions is shown in table below:

Mnemonics	**Meaning**	**Format**	**Operation**	**Flags affected**
MUL	Multiply	MUL S	AL*S8 ⟶ AX AX*S16 ⟶ DX AX	CF, OF AC, SF, PF, ZF undefined
IMUL	Integer multiply	IMUL S	AL*S8 ⟶ AX AX*S16 ⟶ DX AX	CF, OF AC, SF, PF, ZF undefined

Where Q stands for Quotient and R stands for reminder.

Examples:

MUL BL ; AX=AL*BL.

IMUL CH ; AX=AL*CH.

MUL WORDPTR[BX+DI+1234h] ; DX AX=AX*The content of M.L. (16-bit) with an offset of (BX+DI+1234h) as the low byte from ((BX+DI+1234h) and the high byte from (BX+DI+1235h).

IMUL BYTEPTR[5000h] ; AX=AL*The content of M.L. (8-bit) with an offset of (5000h).

MUL CX ; DX AX=AX*CX.

Important note: the **MUL** and **IMUL** instructions do not take the *immediate addressing mode*.

Now to understand the difference between MUL and IMUL instruction, first load **IMUL AH** and put FFFFh in register AX. As a signed byte, FF= -1 so (-1 * -1 = +1). Trace the instruction and see that AX=0001h. now load **MUL AH** and again set AX=FFFFh. Now FF=255, so

255*255=65025 or in Hex. FF*FF=FE01h. tracing this instruction gives AX=FE01h, as expected. Only for byte factors less than 128 do MUL and IMUL give the same results.

Example 4: write an A.L.P. that multiplies two 16-bit numbers stored in BX and CX and store the result in M.L.s specified by [2342h]?

Solution:

```
MOV AX, BX
MUL CX
MOV [2342h], AX
MOV [2344h], DX
HLT
```

H.W. 1: write an A.L.P. that multiplies two 16-bit numbers stored in 2 consecutive M.L.s specified by [BX] then store the result in the next 4 M.L.s followed the 2 consecutive numbers.

3.2.2.4 Division instructions: DIV and IDIV

The **DIV** (for unsigned numbers) and **IDIV** (for signed numbers) instructions are used to perform the division operation between two operands: the *source* is external and the *destination* is internal. The source may be either register or memory with data either 8-bit or 16-bit. The internal destination could be either **AX** for an 8-bit source operand or the registers pair **DX AX** for a 16-bit source operand. This operation is explained in table below:

Division (DIV or IDIV)	**Internal operand**	**External operand (source)**	**Result**
Word / Byte	AX	Register or BYTEPTR of memory	AX / source = AX Q (AL) R(AH)
DWord / Word	DX AX	Register or WORDPTR of memory	(DX AX) / source = DX AX Q (AX) R (DX)

The general format for the multiplication instructions is shown in table below:

Mnemonics	Meaning	Format	Operation	Flags affected
DIV	Divide	DIV S	AX/S8 ⟶ Q(AL) R(AH) DX AX/S16 ⟶ Q(AX) R(DX)	CF, OF, AC, SF, PF, ZF undefined
IDIV	Integer divide	IDIV S	AX/S8 ⟶ Q(AL) R(AH) DX AX/S16 ⟶ Q(AX) R(DX)	CF, OF, AC, SF, PF, ZF undefined

Examples:

DIV BL ; AX/BL ⟶ Q(AL) and R(AH).

IDIV CH ; AX/CH ⟶ Q(AL) and R(AH) .

DIV WORDPTR[BX+DI+1234h] ; (DX AX)/(The content of M.L. (16-bit) with an offset of (BX+DI+1234h) as the low byte from ((BX+DI+1234h) and the high byte from (BX+DI+1235h)) ⟶ The Q(AX) and R(DX).

IDIV BYTEPTR[5000h] ; AX/(The content of M.L. (8-bit) with an offset of (5000h)) ⟶ the Q(AL) and R(AH).

DIV CX ; (DX AX)/CX ⟶ The Q(AX) and R(DX).

Important note: the **DIV** and **IDIV** instructions do not take the *immediate addressing mode*.

The division operation can result in two errors:

1. Division by zero.
2. Division overflow when a large 16-bit no. is divide by a small 8-bit no.. for example, the division of 4000h by 04h, because the result of an 8-bit division must be fill in AL, while the result of the last operation is 1000h that does not fit into AL.

To divide an 8-bit dividend by and an 8-bit divisor by extending the sign bit of AL to fill all bits of AH. This can be done automatically by executing the instruction (**CBW**).

In a similar way, a 16-bit dividend in AX can be divided by 16-bit divisor. In this case the sign bit in AX is extended to fill all bits of DX. The instruction **(CWD)** performs this operation automatically.

Note that **CBW** extends 8-bit in AL to 16 bit in AX while the value in AX will be equivalent to the vale in AL. similarly, **CWD** converts the value in AX to 32-bit in (DX AX) without changing the original value. The general format of thes instructions is shown in table below:

Mnemonics	Meaning	Format	Operation	Flags affected
CBW	Convert byte to word	CBW	**Copy** the MSB of AL to **all bits** of AH	None
CWD	Convert word to double word	CWD	**Copy** the MSB of AX to **all bits** of DX	None

Example 4: write an A.L.P. to divide the 40 by 3 then explain its operation.

Solution:

```
MOV AX, 0028h        ; AX=0028h
MOV BL, 03h          ; BL=03h
DIV BL               ; the explaining is below.
HLT
```

The Q of the division is 13 and in the Hex. it is D so AL=0Dh

The R of the division is 1 so AH=01.

The final result in AX=010Dh.

Example 5: what is the value of AX after executing the following instructions:

```
MOV AX, 0000h
MOV AL, FBh          ; AX=00FBh
CBW
```

Since the MSB of AL=1 then, all bits of AH=1's. So the value of AH=FFh. Then AX=FFFBh

Example 6: what is the value of AX and DX after executing the following instructions:

```
MOV DX, 0000h
MOV AL, FFFBh        ; DX AX = 0000  FFFB
CBW
```

Since the MSB of AX=1 then, all bits of DX=1's. So the value of DX AX= FFFF FFFB.

H.W. 2: trace the following program:

MOV AX, 0324h

MOV BX, 0203h

MUL BL

DIV BH

CWD

HLT

Assume all flags are initially zeros

H.W. 3: write an A.L.P. to express the following equation:

$$\mathbf{X= (2*Y+ \frac{Z}{5}) - W^2}$$

Where X is a M.L. with an offset of 0500h. Assume Y=33h, Z=50h, and W=05h.

3.2.3 Logical Instructions

Logical instructions are bitwise instructions operating on the individual bits. Typical logical operations include logical complement (**NOT**), logical and (**AND**), logical or (**OR**), and logical exclusive or (**XOR**).

i. OR X,Y ; X = X + Y (X **OR** Y).

The truth table of **OR** gate can be shown below:

X	Y	O / P
0	0	0
0	1	1
1	0	1
1	1	1

Examples:

OR AX, BX

OR CX, 30h

OR BL, DH

OR BYTEPTR[5000h], 7Fh

Important note: Here we can use **OR** for ***setting bits*** of any data by putting **1's** against the bits that want to be set & the others are **0's**.

Example 7: Set the 2'nd bit of reg. AL:

AL = x x x x x x x x

02 = 0 0 0 0 0 0 1 0

we put 1 in the position of the 2'nd bit

So the solution will be:

OR AL, 02h

ii. AND X,Y ; X = X . Y (X **AND** Y).

The truth table of **AND** gate can be shown below:

X	Y	O / P
0	0	0
0	1	0
1	0	0
1	1	1

Examples:

AND AX, BX

AND CX, 30h

AND BL, DH

AND BYTEPTR[5000h], 7Fh

Important note: Here we can use **AND** for ***resetting(clearing) bits*** of any data by putting **0's** against the bits that want to be reset & the others are **1's**.

Example 8: Reset the 4'th and 11'th bits of reg. CX:

CX = x x x x x x x x x x x x x x x x

FBF7 = 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1

we put 0's in the positions of 4'th and 11'th bits

So the solution will be:

AND CX,FBF7h

iii. XOR X,Y ; X = X ⊕ Y (X **XOR** Y).

The truth table of **XOR** gate can be shown below:

X	**Y**	**O / P**
0	0	0
0	1	1
1	0	1
1	1	0

Examples:

XOR AX, BX

XOR CX, 30h

XOR BL, DH

XOR WORDPTR[BX+SI+20h], 756Fh

Important note: Here we can use **XOR** for ***inverting bits*** of any data by putting **1's** against the bits that want to be inverted and the others are **0's**.

Example 9: Invert the 1'st & the 8'th bits of reg. DL:

DL = x x x x x x x x

81h = 1 0 0 0 0 0 0 1

we put 1's in the positions of 1'st & 8'th bits

So the solution will be:

XOR DL,81h

iv. NOT X ; Take the 1's complement of X.

X	O / P
0	1
1	0

Examples:

NOT SI ; invert all bits of reg. SI

NOT BL ; invert all bits of reg. BL

NOT WORDPTR[SI+1234h] ; invert all bits of the content of the 16-bit of the memory locations specified by [SI+1234] as the low byte and [SI+1235] as the high byte.

Example 10: what is the value of AL after executing the following instruction:

MOV AL, C3h ; AL = 1100 0011 b

NOT AL

AL = 0011 1100 b = 3Ch

Example 11: Rebuild the following instruction without using OR instruction.

OR AL,BL .

We can rebuild the instruction by using **Boolean Algebra** for the expression of or gate:

$$O/P = \overline{AL + BL} = \overline{\overline{AL} \cdot \overline{BL}}$$

So the solution will be:

```
NOT AL
NOT BL
AND AL,BL
NOT AL
```

Important note: the **NOT** instruction does not take the *immediate addressing mode*.

The general formats for the logical instructions are shown in table below:

Mnemonics	Meaning	Format	Operation	Flags affected
OR	Logical OR	OR D, S	(D) . (S) ⟶ (D)	CF, OF, SF, PF, ZF AC undefined
AND	Logical AND	AND D, S	(D) + (S) ⟶ (D)	CF, OF, SF, PF, ZF AC undefined
XOR	Logical Exclusive OR	XOR D, S	(D) ⊕ (S) ⟶ (D)	CF, OF, SF, PF, ZF AC undefined
NOT	Logical NOT	NOT D	(D) ⟶ $\overline{(D)}$	

H.W. 1: trace the following program:

```
MOV AX, 0324h
MOV BX, 0203h
OR AX, BX
AND BX, 3C2F
XOR AX, DCAB
NOT BH
HLT
```

Assume all flags (Z, S, P) are initially zeros

H.W. 2: write an A.L.P. to express the following logical expression:

$X = \overline{(Y \oplus Z) \cdot (W+55)}$

Where X is a M.L. with an offset of 0500h. Assume Y=33h, Z=50h, and W=05h.

H.W. 3: write an instructions that do the following:

Rebuild

i. **XOR AX, BX**
ii. **AND CX, SI**
iii. **XOR AX, FF**
iv. **NOT BYTEPTR [3500h]**

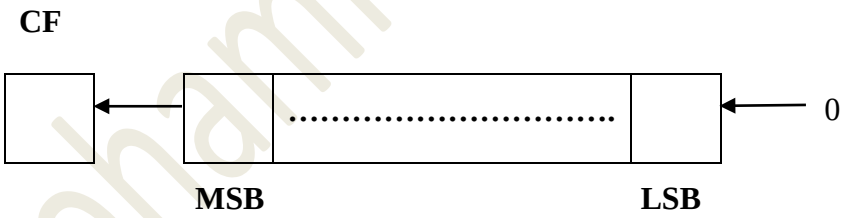
3.2.4 Shifting and Rotating instructions

3.2.4.1 Shifting instructions

i. Logical Shifting

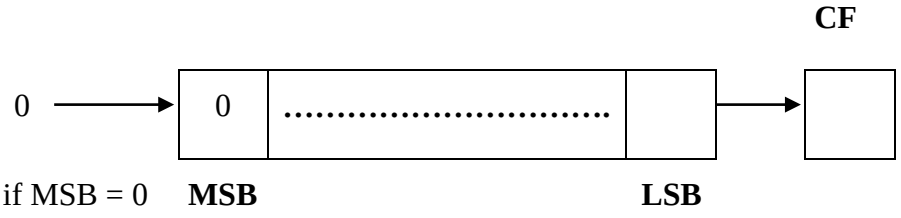
1. **SHR** op1,op2 . shift right op1(logical) by op2. The shift to the right can be done by enter 0 from the left and every bit jump to the other bit to the right & the (**LSB**) from right will be transferred to the carry flag.

2. **SHL** op1,op2 . shift left op1(logical) by op2. The shift to the left can be done by enter 0 from the right and every bit jump to the other bit to the left & the (**MSB**) will be transferred to the carry flag.

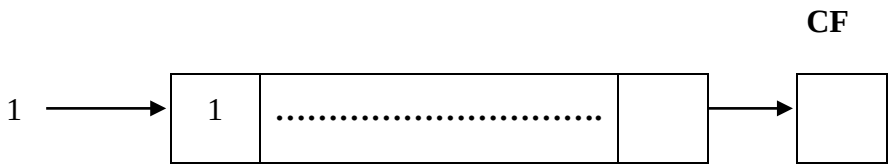


ii. Arithmetic Shifting

1. **SAR** op1,op2 . shift right (arithmetic). This instruction check the sign bit (**MSB**) & see if 0 enter zero from left if 1 enter one from left & keep the sign bit. i.e. copy the sign bit as many times as shift & the (**LSB**) transferred to the carry flag.

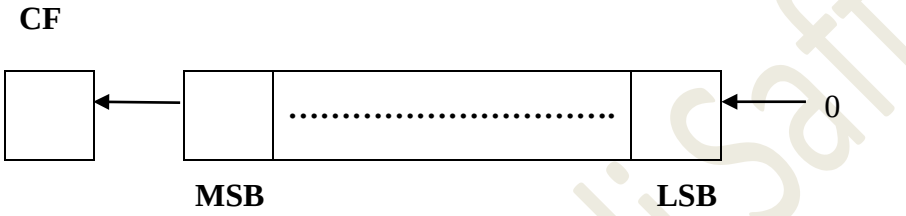


Enter zero if MSB = 0



Enter one if MSB = 1

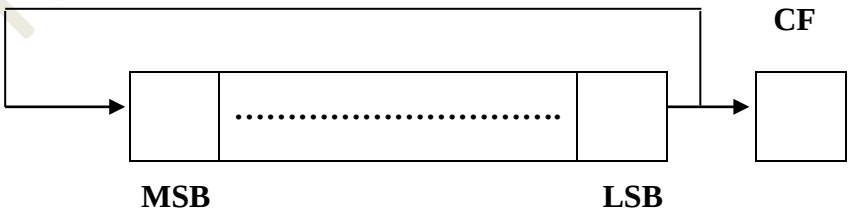
2. **SAL** op1,op2 . shift left op1(arithmetic) by op2. The shift to the left can be done by enter 0 from the right and every bit jump to the other bit to the left & the (**MSB**) will be transferred to the carry flag.



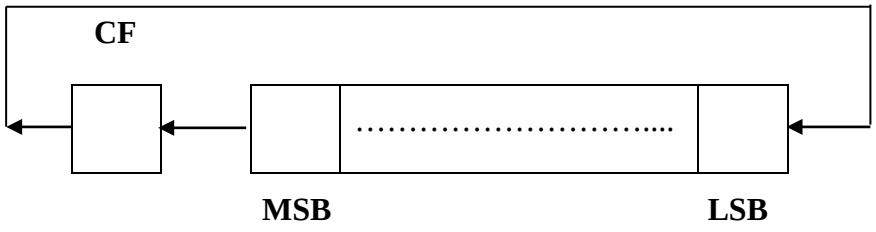
3.2.4.2 Rotating instructions

1. **ROL** op1, op2. Rotate data from right to left & the last bit (**MSB**) rotated from left transferred to carry flag.

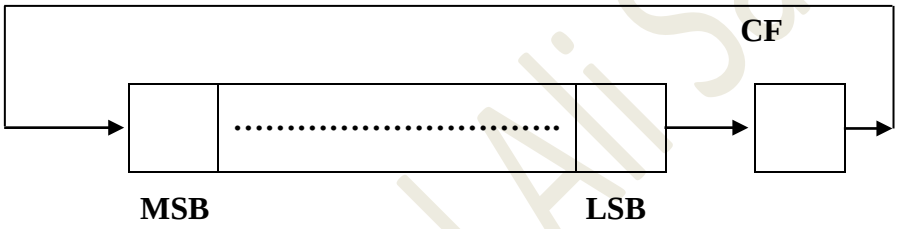
2. **ROR** op1, op2. Rotate data from left to right & the last bit (**LSB**) rotated from right transferred to carry flag.



3. **RCL** op1, op2. Rotate data from right to left through carry & the old value of the carry transferred to the (**LSB**).



4. **RCR** op1, op2. Rotate data from left to right through carry & the old value of the carry transferred to the (**MSB**).



Example 12: Let AH = 1000 0000 & CF = 0, what is the result of AH & CF after executing the following instructions?

INSTRUCTION	RESULT	CF
SHL AH,1		
SHR AH,1		
SAL AH,1		
SAR AH,1		
ROL AH,1		
ROR AH,1		
RCL AH,1		
RCR AH,1		

Important Notes:

1. ***All*** rotation & shifting operations change the carry flag.
2. If we want to rotate or shift data for more than one time we must use **CL** register for this purpose.

   Example 13: Rotate reg. **AX** 6 times to the left without carry:

   MOV CL,06H

   ROL AX,CL

3. every **n** times shift to the right represent ***divide*** by 2^n.

   Example 14: Let AL = 0000 1000 = 08H

   SHR AL,1 AL = 0000 0100 = 04H

   SHR AL,1 AL = 0000 0010 = 02H

   SHR AL,1 AL = 0000 0001 = 01H

4. Every **n** times shift to the left represent ***multiply*** by 2^n.

   Example 15: Let AL = 0001 0000 = 10H

   SHL AL,1 AL = 0010 0000 = 20H

   SHL AL,1 AL = 0100 0000 = 40H

   SHL AL,1 AL = 1000 0000 = 80H

5. The **SHL**, **SAL** & **SHR** cause of losing data.

   Example 16: Let BX = FFFFH

   So after executing the following instructions:

   MOV CL,10H

   SHR BX,CL

   BX = 0000H

6. The **SAR** cause either losing data or make all bits of data = 1

   Example 17: Let AL = 0100 0100 = 44H

MOV CL,07

SAR AL,CL

Since **MSB** = 0 the result will be AL = 0000 0000 = 00H.

Example 18: Let AL = 1001 0110 = 96H

MOV CL,07

SAR AL,CL

Since **MSB** = 1 the result will be AL = 1111 1111 = FFH.

H.W. :Write a set of instructions to perform the following operations:

- **7AL without using MUL instruction. (AL = 05H).**
- **Copy the value of 3rd bit in M.L. [2000] to all bits of reg. BH.**

Example 19: Write an A.L.P. to copy the value of the 3rd bit of the memory location specified by [BP+DI+33h] to all bits of register BX.

Sol.

```
MOV BX, [BP+DI+33h]
MOV CL, 0Dh
SHL BX, CL
MOV CL, 0Fh
SAR CX, CL
HLT
```

Example 20: Rebuild CBW.

Sol.

```
MOV AH, AL
MOV CL, 07
SAR AH, CL
HLT
```

Example 21: Write an A.L.P. to rotate a 32-bit of data stored into DX AX registers to the left 1 time.

Sol.

```
ROL DX, 1
RCL AX, 1
ROR DX, 1
RCL DX, 1
HLT
```

H.W. 2: Write an A.L.P. to copy the value of the carry flag to all bits of a M.L. specified by BX.

H.W. 3: Write an A.L.P. to rotate a 32-bit of data stored into DX AX registers to the right 1 time.

H.W. 4: Rebuild CWD.

3.2.5 Flag Control Instructions

The instruction set includes a group of instructions that when executed directly affects the state of the flags. These instructions are shown in table below:

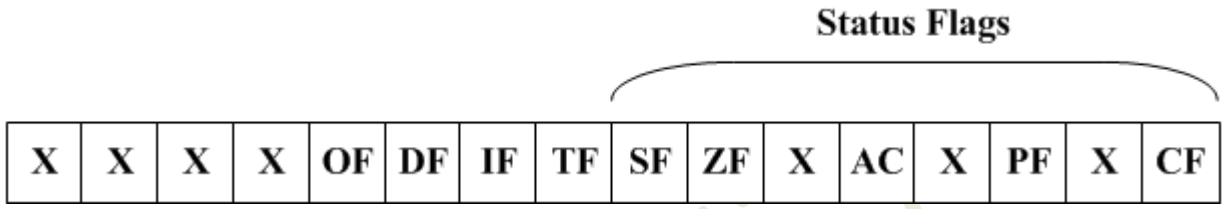
Mnemonics	Meaning	Format	Operation	Flags affected
CLC	Clear carry flag	CLC	**CF = 0**	CF
STC	Set carry flag	STC	**CF = 1**	CF
CMC	Complement carry flag	CMC	**CF = $\overline{\text{CF}}$**	CF
CLD	Clear direction flag	CLD	**DF = 0**	DF
STD	Set direction flag	STD	**DF = 1**	DF
CLI	Clear interrupt flag	CLI	**IF = 0**	IF
STI	Set interrupt flag	STI	**IF = 1**	IF

LAHF and SAHF instructions

The LAHF and SAHF instructions are seldom used because they were designed as bridge instructions. These instructions allowed 8085 (an early 8-bit microprocessor) software to be translated into 8086 software by a translation program. Because any software that required translation was completed many years ago, these instructions have little application today. The

LAHF instruction transfers the rightmost 8 bits of the flag register (status flags) into the AH register. The **SAHF** instruction transfers the AH register into the status flags of the flag register.

Mnemonics	Meaning	Format	Operation	Flags affected
LAHF	Load AH reg. from status flags	LAHF	**AH = low byte of flag reg.**	none
SAHF	Store AH reg. in status flags	SAHF	**Low byte of flag reg. = AH**	SF,ZF,ACF,PF,CF



Figutre Flag Register

Example 22: Write an A.L.P. to square the value of the right byte of the flag register (status flags).

Sol.

```
LAHF
MOV AL,AH
MUL AL        or    MUL AH
SAHF
HLT
```

3.2.6 Program Flow Control Instructions

The purpose of a jump instruction is to alter the execution path of instructions in the program. The code segment register and instruction pointer keep track of the next instruction to be fetched for execution.so a jump ivolves altering the contents of either the IP register or CS and IP registers together. In this way an execution continues at an address other than the next sequential instruction.

There are two types of jump instructions: unconditional jump and conditional jump.

Unconditional Jump

This type of jump instruction (**JMP**) allows the programmer to skip sections of a program and branch to any part of the memory for the next instruction. In an unconditional jump, no status requirements are imposed for the jump to occur. That is, as the instruction is executed, the jump always takes place to change the execution sequence.

The general format of this instruction is shown below

Mnemonics	Meaning	Format	Operation	Flags affected
Jmp	Unconditional jump	Jump operand	Jump to the address specified by operand	none

The destination (target) operand specifies the addres of the instruction being jump to. This operand can be immediate value, a general-purpose register, or a memory location, according to this, the **JMP** instruction can be classified into:

a. **Intrasegment:** is limited with addresses within the current code segment.
b. **Intersegment:** permits jumps from one code segment to another.

***Important Note*:** Jump instructions specified with a **Short-Label, Near-Label, Memptr 16** or **Regptr16** represent **intrasegment** jumps while jump instructions specified with a **far-label** and **Memptr32** represent intersegment jumps.
According to that, the unconditional jumps can be classified into these types:

i. **Short Jump:** is a 2-byte instruction that allows jumps or branches to memory locations within +127 and -128 bytes from the memory location following the jump.

Ex. **JMP** 32h

$\mathbf{IP}_{new}=\mathbf{IP}_{old}+32h$

Next address= $\mathbf{CS*10}+\mathbf{IP}_{new}$

JMP	Displacement
Byte 1	Byte 2

ii. **Near Jump:** the 3-byte near jump allows a branch or jump within ±32K bytes (anywhere) from the instruction in the current code segment.

JMP 1234h

$\mathbf{IP}_{new}=\mathbf{IP}_{old}+1234h$

Next address= $\mathbf{CS*10}+\mathbf{IP}_{new}$

JMP	Disp. High	Disp. Low
Byte 1	Byte 2	Byte 3

iii. **Far Jump:** the far jump obtains a new segment and offset address to accomplish the jump. Notice that the operand (32-bit) specified with the jump is a new values to the **IP** and **CS** and not a displacement as in short and near jumps.

JMP 1456:F2E6h

After executing the instruction shown above, the following values will be accomplished:

IP=F2E6h CS=1456h

Next address=$\mathbf{CS*10}+\mathbf{IP}_{new}$

iv. **Reg16-bit** or **Mem16-bit:** the unconditional jump address can also be specified indirectly by the contents of a register or memory location.

JMP BX ; $\mathbf{IP}_{new}$=BX

JMP [BX] ; $\mathbf{IP}_{new}\mathbf{(low)}$=[BX]

; $\mathbf{IP}_{new}\mathbf{(High)}$=[BX+1]

v. **Mem32-bit:**as using far jump here an indirect way is used to specify the offset and code segment address and that done by using **Mem32-bit** operand (four consecutive memory bytes starting at specified address).

```
JMP DWORD [BX]          ; IP-low=[BX]
                        ; IP-high=[BX+1]
                        ; CS-low=[BX+2]
                        ; CS-high=[BX+3]
```

Conditional Jump

The conditional jump depends on the values of the status flags i.e. (**S,Z,O,P,C**). for this type of jump, the jump does not happen if the condition that affect the status flags does not exist. Notice that these instructions are ***short*** jump instructions and this limits the jump within +127 and -128 bytes from the loction following the conditional jump. A list of each of the conditional jump instructions is shown in the following table:

Mnemonics	Meaning	Condition
JA	Above	CF=0 and ZF=0
JAE	Above or Equal	CF=0
JB	Belo	CF=1
JBE	Belo or Equal	CF=1 or ZF=1
JC	Carry	CF=1
JCXZ	CX register is zero	CF=0 or ZF=0
JE	Equal	ZF=1
JG	Greater	ZF=0 and SF=OF
JGE	Greater or Equal	SF=OF
JL	Less	(SF xor OF)=1
JLE	Less or Equal	((SF xor OF) or ZF)=1
JNA	Not Above	CF=1 or ZF=1
JNAE	Not Above nor Equal	CF=1
JNB	Not Below	CF=0

JNBE	Not Below nor Equal	CF=0 and ZF=0
JNC	No Carry	CF=0
JNE	Not Equal	ZF=0
JNG	Not Greater	((SF xor OF) or ZF)=1
JNGE	Not Greater nor Equal	(SF xor OF)=1
JNL	Not Less	SF=OF
JNLE	Not Less nor Equal	ZF=0 and SF=OF
JNO	No Overflow	OF=0
JNP	No Parity	PF=0
JNS	No Sign	SF=0
JNZ	No Zero	Zf=0
JO	Overflow	OF=1
JP	Parity	PF=1
JPE	Parity Even	PF=1
JPO	Parity Odd	PF=0
JS	Sign	SF=1
JZ	Zero	ZF=1

Important notes:

- The terms Greater than and Less than refer to ***signed numbers***, therefore when signed numbers are compared; the **JG, JL, JGE, JLE, JE,** and **JNE** are used.
- The terms Above and Below refer to ***unsigned*** numbers, therefore when unsigned numbers are compared use **JA, JB, JAE, JBE, JE,** and **JNE** are used.

According to the nature of there working, conditional jumps can be classified as:

i. If statement:

As mentioned before, the conditional jumps do not exist if the condition does not exist.

If Condition is existed } the condition depends on the type of the conditional

Jump to LABEL jump

Continue program

JMP END

LABEL: certain instructions

END: HLT

Ex.1:

```
MOV AX,2500h
MOV BX,2006h
ADD AX,BX
JP *
MOV CL,02h
JMP END
*: MOV CL,01h
END: HLT
```

Ex.2:

```
MOV AX,5454h
CMP AL,AH
JZ *
MOV BYTEPTR[3000h],01h
JMP END
*: MOV BYTEPTR[3000h],02
END HLT
```

Ex.3: Write an A.L.P. to check the contents of a M.L. specified by [SI] if it is below or equal (50h) then put (01h) in reg. BL. Else put (02h) in reg. BL.

Sol.

```
CMP BYTEPTR[SI],50h
JBE *
MOV BL,02h
JMP END
*: MOV BL,01h
END: HLT
```

EX. 4: Write an A.L.P. to check the number stored in a M.L. specified by [BX+DI+2673h] if it is +ve put (01h) in reg. CH, if it is equal zero put (02h) in reg. CH, else put (03h) in reg. CH.

Sol.

```
CMP BYTEPTR[BX+DI+2673],00H
JE *
JA **
MOV CH,03
JMP END
*: MOV CH,01H
JMP END
**: MOV CH,02
END HLT
```

ii. Loop Program:

In computer programming, a loop is a sequence of instructions that is continually repeated until a certain condition is reached. Typically, a certain process is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number. If it hasn't, the next instruction in the sequence is an instruction to return to the first instruction in the sequence and repeat the sequence. If the condition has been reached, the next instruction 'falls through' to the next sequential instruction or branches outside the loop. A loop is a fundamental programming idea that is commonly used in writing programs.

There are two types of loop programming structure. These are:

i. Descending Loop:

This loop starts with the ***final*** value of the counter then ***decremented*** by one and check whether the counter equal zero or not. If it is not zero then, return and continue the program. If it is reached zero, it will end the loop and continue the program.

Ex. From C++ a descending loop is like the following:

```
for (i=10; i > 0; i--)
```

In assembly language ***in general***, the **CX** and **CL** registers are used for the counter instead of the variables in the loop programs.

```
MOV CL,0Ah          or    MOV CX, 000Ah
.
*
.
.
DEC CL
JNZ *
.
(CONTINUE PROGRAM)
.
HLT
```

ii. Ascending Loop:

This loop starts with the ***least*** value of the counter then ***incremented*** by one and ***copamre*** whether the counter reaches its final value or not. If it is not reached then, return to the loop program and continue. If it is equal to the final value then, end the loop and continue the program.

Ex. From C++ a descending loop is like the following:

for (i=1; i ≤ 15; i++)

Again, in assembly language ***in general***, the **CX** and **CL** registers are used for the counter instead of the variables in the loop programs.

```
MOV CL,01h          or    MOV CX, 0001h
.
*
.
.
INC CL
CMP CL, 0Fh
JNZ *
.
(CONTINUE PROGRAM)
.
HLT
```

Ex. 4: Write an A.L.P. to find the average of a student of 8 degrees. The degrees stored in an M.L. starts at [5000h].

Sol.

```
MOV CL, 08h          ; Initialize the counter.
MOV AL, 00h          ; The summation of the degrees must equal zero.
MOV AH, 00h
MOV DI, 5000         ; Put the value of the 1st address in (DI or SI or BX)
                       i.e. use (DI, SI, and BX) as a pointer to the memory.
*: ADC AL, BYTEPTR[DI]    ; Summation for the degrees.
DEC CL
JNZ *
```

MOV BL, 08h
DIV BL ; calculate the average by dividing the summation
HLT which is now in (AX) by number of degrees (8)
degrees

Ex. 5: Write an A.L.P. to move a block of **12** bytes of data starting at offset address [3500h] to another block starting at offset address [7080h]. Assume that both blocks are in the same segment whose starting address is 1234h.

Sol.

MOV AX,1234h
MOV DS, AX
MOV SI, 3500h
MOV DI, 7080h
MOV CL, 0Ch
#: MOV DL, [SI]
MOV [DI], DL
INC SI
INC DI
DEC CL
JNZ #
HLT

H.W. 1: Write an A.L.P. that multiplicities a block of 20 bytes (vector of M.L.s) of data starts at an offset address [4000h] with another vector of M.L.s starts at an offset address of [8858h] then put the results in a third vector starts [8000h]. Assume that all of the three vectors are in the same segment that's its start address is DATASEGM.

Subroutines:

Subroutine is a sub program needed to perform a particular sub-task many times on different data values. Instead of putting this program in the main program, subroutine can be stored in a specified M.L. of the memory and called it using a **CALL** instruction. Whenever this sub-program needed to be executed, there is no need to write this program again and again, just call it and execute it.

When the program called a subroutine, executed it then, it should return to the main program. This operation can be done by putting the **RET** instruction at the end of the subroutine.

CALL Instruction:

The CALL instruction is a special branch instruction that performs the following operations:

- Push the contents of the next instruction address (IP) on the top of the stack.
- Update the stack pointer (SP).
- Branch to the target address specified by the CALL instruction.

Important Note: The CALL instruction is similar to the JMP instruction, that it is intrasegment and intersegment. The general format of the CALL instruction is shown in the following table:

Mnemonics	Meaning	Format	Operation	Flags affected
CALL	Subroutine Call	CALL Operand	Execution continues from the address of the subroutine specified by the operand.	None

Operand may be one of the: Near, Far, REG16, Mem16, Mem32).

Near Call: Execution of a near CALL causes the contents of the IP to be saved in the stack, and a new 16-bit value which corresponds to the address of the first subroutine instruction to be into IP register.

e.g. **CALL** 1453h

Far CALL: Execution of Far CALL causes the contents of IP and CS registers to be stored in the stack respectively and a new 32-bit value for code segment and offset to be loaded into IP and CS registers.

e.g. **CALL** 1781:2712h

Indirect CALL: (Reg16, Mem16, Mem32).

e.g.s **CALL** BX

CALL [BX]

RET Instruction:

The Return instruction (RET) is a special branch instruction that performs the following operations:

- Pop the return address from the top of the stack.
- Update the stack pointer (SP).

The RET instruction removes the 16-bit number (near return) from the stack and places it into IP, or removes a 32-bit number. (far return) and places it into IP and CS.

Subroutine Structure:

Main program

.

.

CALL SUB

Continue program

HLT

SUB.

Program instructions

.

.

RET

Ex.6: repeat Ex.5 using subroutine.

Sol.

MOV AX,1234h
MOV DS, AX
MOV SI, 3500h
MOV DI, 7080h
MOV CL, 0Ch
#:CALL Copy
INC SI
INC DI
DEC CL
JNZ #
HLT

Copy: MOV DL, [SI]
MOV [DI], DL
RET

H.W. 2: Repeat H.W. 1 using subroutine.

The Stack

The stack is a temporary storage originally used to preserve return addresses when a subroutine is called. It is so convenient for temporary storage that it is used to save the contents of certain registers or some other main program parameters.

The instruction that is used to save parameters on the stack is **PUSH** instruction and the instruction that is used to retrieve them from the stack is the **POP** instruction. The standard formats of these instructions are shown in the following table:

Mnemonics	Meaning	Format	Operation	Flags affected
PUSH	Save word onto stack	PUSH S	S ⟶ M.L. specified by SP	None
POP	Retrieve word off stack	POP D	M.L. specified by SP ⟶ D	None

Important Note:

- The allowed operands are (general purpose registers, a segment register (excluding CS), or a storage location in memory).
- PUSH and POP instructions always store or retrieve ***WORDS*** of data only.

The work of stack is organized in the form of First In Last Out (FILO) or Last In First Out (LIFO).

Ex.s:

PUSH AX

PUSH [BX+50h]

POP DS

POP [SI]

Offsets of The Stack

i. SP register

Represents the ***top of the stack*** and it is affected by every PUSH and POP operations.

For PUSH instruction the SP register will be ***decremented*** by **2**.

$SP_{new} = SP_{old} - 2$

Ex.s: Assume SP=5006h, DX=F736h.

PUSH DX ; DH ⟶ (SP-1)

; DL ⟶ (SP-2)

	XX	[5002h]
	XX	[5003h]
DL	36	[5004h]
DH	F7	[5005h]
Top of stack (SP)	XX	[5006h]

The new value of the SP will be: SP=5004h.

For POP instruction the SP register will be ***incremented*** by **2**.

$SP_{new} = SP_{old} + 2$

Ex.s: Assume SS=3000h, SP=7000h.

POP AX ; (SP+1) ⟶ AH

; (SP+2) ⟶ AL

PA=SS*10h+SP

= 3000*10+7000h

=37000h

Top of stack (SP)	XX	[37000h]
AH ⟵	55	[37001h]
AL ⟵	36	[37002h]
	XX	[37003h]
	XX	[37004h]

AX=5536h

SP=7002h

ii. BP register

If anyone wants to reach a certain memory location in the stack (i.e. not the top of the stack) then, the BP register will be used. The BP used for the ***random access*** of a certain M.L. in the stack.

Ex. 1: Write A.L.P. to put 507Dh in register CX. Assume the top of the stack is 6040h in a segment of memory which its start is 15DDh using stack operations.

Sol.

```
MOV AX, 15DDh
MOV SS, AX
MOV SP, 6040h
MOV AX, 507Dh
PUSH AX
POP CX
HLT
```

Ex. 2: You have AX=1234h, BX=3200h, SP=120Ah. What are the values of AX, BX CX, DX, and SP after executing the following program:

```
PUSH AX
PUSH BX
POP CX
PUSH BX
POP DX
HLT
```

AX=1234h BX=3200h CX=3200h DX=3200 SP=1208h.

Ex. 3: Write an A.L.P. to store the following sequence in the stack: 0, 1, 4, 5. Assume the start of the stack segment and offset is 1000:0500.

Sol.

```
   MOV AX, 1000h
   MOV SS, AX
   MOV SS, 0500h
   MOV BX, 0000h
   MOV CL, 02h
   ROR BX, CL
   JC *
   ROL BX, CL
   PUSH BX
*: INC BX
   CMP BX, 0006h
   JNZ #
   HLT
```

3.2.7 String and String-Handling Instructions

The 8086 microprocessor is equipped with special instructions to handle string operations. String means a series of data words (or bytes) that reside in consecutive memory locations. There are many string instructions in the instruction set of 8086, four of these instructions are discussed here.

1. <u>MOVSB, MOVSW</u>:

An element specified by the (SI) reg. in the current data segment (DS) reg. is moved to the locations specified by the (DI) reg. in the current (ES) reg. . The move can be performed on a byte or a word of data.

MOVSB (8-bit) (1 byte)

This instruction will move a byte of data from the M.L. in ***data segment*** specified with **SI** as the ***source*** of data to a M.L. in ***extra segment*** specified with **DI** as the **destination** for data.

MOVSB ; move byte from (DS:SI) to (ES:DI) .

For DF = 0,

SI = SI + 1 , DI = DI + 1

For DF = 1,

SI = SI – 1 , DI = DI – 1

MOVSW (16-bit) (2 byte)

This instruction will move a word of data from the M.L. in ***data segment*** specified with **SI** as the ***source*** of data to a M.L. in ***extra segment*** specified with **DI** as the **destination** for data.

MOVSW ; move word from (DS:SI) to (ES:DI) & (DS: SI + 1) to (ES:DI + 1).

For DF = 0,

SI = SI + 2 , DI = DI + 2

For DF = 1,

SI = SI – 2 , DI = DI – 2

2. <u>CMPSB, CMPSW</u>:

These instructions can be used to compare two elements in the same or different strings. It subtracts the content of (ES : DI) from the content of (DS : SI) and adjusts flags (CF, PF, AF, ZE, SF, & OF). The result of subtraction is not saved but it affects on the flag reg. (FX) only.

CMPSB ; compare the string content byte in (DS:SI) with the content of (ES:DI) .

For DF = 0,

SI = SI + 1 , DI = DI + 1

For DF = 1,

SI = SI – 1 , DI = DI – 1

CMPSW ; compare the string content word in (DS:SI) with the content of (ES:DI) and (DS:SI+1) with the content of (ES:DI+1).

For DF = 0,

SI = SI + 2 , DI = DI + 2

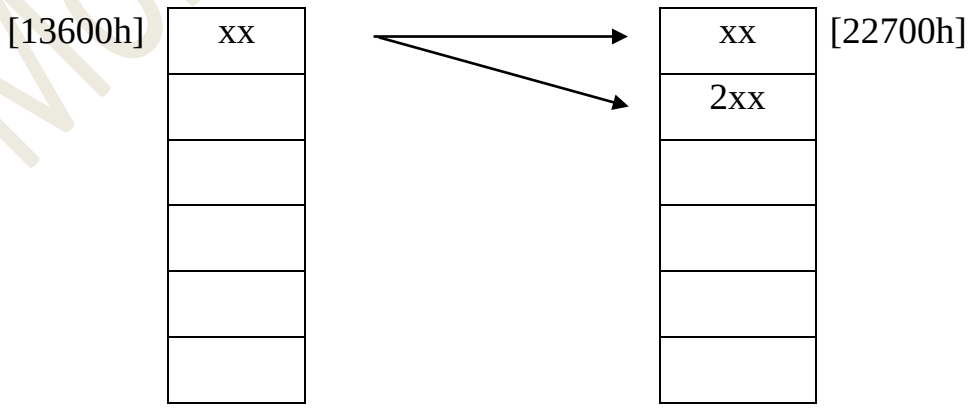
For DF = 1,

SI = SI – 2 , DI = DI – 2

3. <u>STD</u>: make DF = 1 .

4. <u>CLD</u>: make DF = 0 .

Ex.1: Write an A.L.P. to perform the following operation:



i.e. move byte from the 1st memory to the 2nd memory and move the double of the same byte to the 2nd memory location of the 2nd vector. The 1st vector of the memory is about 8 elements.

Ex.2: Write an A.L.P. to find the number of students that had the same marks in two exams. The marks of the first exam is stored in data segment starting at offset 1000. While the marks of the second exam is stored in extra segment starting at offset 2000. NOTE DS = 2500 and ES = 3000. Total number of students is 50 (decimal).