8086 ASSEMBLY LANGUAGE PROGRAMMING

Contents at a glance:

- ✓ 8086 Instruction Set
- ✓ Assembler directives
- ✓ Procedures and macros.

8086 MEMORY INTERFACING:

- ✓ 8086 addressing and address decoding
- ✓ Interfacing RAM, ROM, EPROM to 8086

INSTRUCTION SET OF 8086

The 8086 instructions are categorized into the following main types

(i) **Data copy /transfer instructions:** These type of instructions are used to transfer data from source operand to destination operand. All the store, load, move, exchange input and output instructions belong to this category.

(ii) **Arithmetic and Logical instructions:** All the instructions performing arithmetic , logical, increment, decrement, compare and ASCII instructions belong to this category.

(iii) **Branch Instructions:** These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instruction belong to this class.

(iv) **Loop instructions:** These instructions can be used to implement unconditional and conditional loops. The LOOP, LOOPNZ , LOOPZ instructions belong to this category.

(v) **Machine control instructions:** These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions belong to this class.

(vi) **Flag manipulation instructions:** All the instructions which directly effect the flag register come under this group of instructions. Instructions like CLD, STD, CLI, STI etc.., belong to this category of instructions.

(vii) **Shift and Rotate instructions:** These instructions involve the bit wise shifting or rotation in either direction with or without a count in CX.

(viii) **String manipulation instructions:** These instructions involve various string manipulation operations like Load, move, scan, compare, store etc..,

1. Data Copy/ Transfer Instructions:

The following instructions come under data copy / transfer instructions:

MOV	**PUSH**	**POP**	**IN**	**OUT**	**PUSHF**	**POPF**	**LEA**	**LDS/LES**	**XLAT**
XCHG	**LAHF**	**SAHF**							

Data Copy/ Transfer Instructions:

MOV: MOVE: This data transfer instruction transfers data from one register / memory location to another register / memory location. The source may be any one of the segment register or other general purpose or special purpose registers or a memory location and another register or memory location may act as destination.

Syntax: 1) MOV mem/reg1, mem/reg2

3) MOV AX, 5000H; Immediate
4) MOV AX, BX; Register
5) MOV AX, [SI]; Indirect
6) MOV AX, [2000H]; Direct
7) MOV AX, 50H[BX]; Based relative, 50H displacement

PUSH: Push to Stack: This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction. The actual current stack-top is always occupied by the previously pushed data. Hence, the push operation decrements SP by two and this store the two-byte contents of the operand onto the stack. The higher byte is pushed first and then the lower byte. Thus out of the two decremental stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address.

Syntax: PUSH reg
[SP] ← [SP]-2
[[S]] ← [reg]

Ex:

1) PUSH AX
2) PUSH DS
3) PUSH [5000H]; content of location 5000H & 5001H in DS are pushed onto the stack.

POP: Pop from stack: This instruction when executed, loads the specified register / memory location with the contents of the memory location of which address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.

Syntax:

i) POP mem
[SP] ← [SP] +2
[mem] ← [[SP]]

ii) POP reg
[SP] ← [SP] + 2
[reg] ← [[SP]]

Ex:

1. POP AX
2. POP DS
3. POP [5000H]

XCHG: Exchange: This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them may be a memory location. However, exchange of data contents of two memory locations is not permitted.

Syntax:

i) XCHG AX, reg 16
[AX] ⟷ [reg 16]
Ex: XCHG AX, DX

ii) XCHG mem, reg
[mem] ⟷ [reg]
Ex: XCHG [BX], DX

Register and memory can be both 8-bit and 16-bit and memory uses DS as segment register.

iii) XCHG reg, reg
[reg] ⟷ [reg]

[mem/reg1] ← [mem/reg2]

Ex: MOV BX, 0210H
MOV AL, BL
MOV [SI], [BX] → is not valid

Memory uses DS as segment register. No memory to memory operation is allowed. It won't affect flag bits in the flag register.

2) MOV mem, data
[mem] ← data

Ex: MOV [BX], 02H
MOV [DI], 1231H

3) MOV reg, data
[reg] ← data

Ex: MOV AL, 11H
MOV CX, 1210H

4) MOV A, mem
[A] ← [mem]

Ex: MOV AL, [SI]
MOV AX, [DI]

5) MOV mem, A
[mem] ← A
A ← : AL/AX

Ex: MOV [SI], AL
MOV [SI], AX

6) MOV segreg,mem/reg
[segreg] ← [mem/reg]

Ex: MOV SS, [SI]

7) MOV mem/reg, segreg
[mem/reg] ← [segreg]

Ex: MOV DX, SS

In the case of immediate addressing mode, a segment register cannot be destination register. In other words, direct loading of the segment registers with immediate data is not permitted. To load the segment registers with immediate data, one will have to load any general-purpose register with the data and then it will have to be moved to that particular segment register.

Ex: Load DS with 5000H
1) MOV DS, 5000H; Not permitted (invalid)

Thus to transfer an immediate data into the segment register, the convert procedure is given below:

2) MOV AX, 5000H
MOV DS, AX

Both the source and destination operands cannot be memory locations (Except for string instructions)

Other MOV instructions examples are given below with the corresponding addressing modes.

Ex: XCHG AL, CL
XCHG DX, BX

Other examples:

1. XCHG [5000H], AX; This instruction exchanges data between AX and a memory location [5000H] in the data segment.
2. XCHG BX; This instruction exchanges data between AX and BX.

I/O Operations:

IN: Input the port: This instruction is used for reading an input port. The address of the input port may be specified in the instruction directly or indirectly AL and AX are the allowed destinations for 8 and 16-bit input operations. DX is the only register (implicit), which is allowed to carry the port address.

Ex: 1. IN AL, DX
[AL] ← [PORT DX]
Input AL with the 8-bit contents of the port addressed by DX

2. IN AX, DX
[AX] ← [PORT DX]
3. IN AL, PORT
[AL]← [PORT]

4. IN AX, PORT
[AX]←[PORT]

5. IN AL, 0300H; This instruction reads data from an 8-bit port whose address is 0300H and stores it in AL.

6. IN AX ; This instruction reads data from a 16-bit port whose address is in DX (implicit) and stores it in AX.

OUT: Output to the Port: This instruction is used for writing to an output port.The address of the output port may be specified in the instruction directly or implicitly in DX. Contents of AX or AL are transferred to a directly or indirectly addressed port after execution of this instruction. The data to an odd addressed port is transferred on D_8 –D_{15} while that to an even addressed port is transferred on D_0-D_7.The registers AL and AX are the allowed source operands for 8-bit and 16-bit operations respectively.

Ex: 1. OUTDX,AL
[PORT DX] ← [AL]
2. OUT DX,AX
[PORT DX] ← [AX]
3. OUT PORT,AL
[PORT] ← [AL]
4. OUT PORT,AX
[PORT] ← [AX]

Output the 8-bit or 16-bit contents of AL or AX into an I/O port addressed by the contents of DX or local port.

5. OUT 0300H,AL; This sends data available in AL to a port whose address is 0300H
6. OUT AX; This sends data available in AX to a port whose address is specified implicitly in DX.

2. Arithmetic Instructions:

ADD	**ADC**	**SUB**	**SBB**	**MUL**	**IMUL**	**DIV**	**IDIV**	**CMP**	**NEGATE**
INC	**DEC**	**DAA**	**DAS**	**AAA**	**AAS**	**AAM**	**AAD**	**CBW**	**CWD**

These instructions usually perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions. The arithmetic instructions affect all the conditional code flags. The operands are either the registers or memory locations immediate data depending upon the addressing mode.

ADD: Addition: This instruction adds an immediate data or contents of a memory location specified in the instruction or a register (source) to the contents of another register (destination) or memory location. The result is in the destination operand. However, both the source and destination operands cannot be memory operands. That means memory to memory addition is not possible. Also the contents of the segment registers cannot be added using this instruction. All the condition code flags are affected depending upon the result.

Syntax:

i. ADD mem/reg1, mem/reg2
[mem/reg1]← [mem/reg2] + [mem/reg2]

Ex : ADD BL, [ST]
ADD AX, BX

ii. ADD mem, data
[mem]←[mem]+data
Ex: ADD Start, 02H
ADD [SI], 0712H

iii. ADD reg, data
[reg]←[reg]+data
Ex: ADD CL, 05H
ADD DX, 0132H

iv. ADD A, data
[A]←[A]+data
Ex: ADD AL, 02H
ADD AX, 1211H

Examples with addressing modes:

1. ADD AX, 0100H	Immediate
2. ADD AX, BX	Register
3. ADD AX, [SI]	Register Indirect
4. ADD AX, [5000H]	Direct
5. ADD [5000H], 0100H	Immediate
6. ADD 0100H	Destination AX (implicit)

ADC: Add with carry: This instruction performs the same operation as ADD instruction, but adds the carry flag bit (which may be set as a result of the previous calculations) to the result. All the condition code flags are affected by this instruction.

Syntax:

i. ADC mem/reg1, mem/reg2
[mem/reg1]←[mem/reg1]+[mem/reg2]+CY

Ex: ADC BL, [SI]
ADC AX, BX

ii. ADC mem,data
[mem]←[mem]+data+CY
Ex: ADC start, 02H
ADC [SI],0712H

iii. ADC reg, data

[reg]←[reg]+data+CY

Ex: ADC AL, 02H

ADC AX, 1211H

Examples with addressing modes:

1. ADC 0100H	Immediate(AX implicit)
2. ADC AX,BX	Register
3. ADC AX,[SI]	Register indirect
4. ADC AX,[5000H]	Direct
5. ADC [5000H],0100H	Immediate

SUB: Subtract: The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand. Source operand may be a register or a memory location, but source and destination operands both must not be memory operands. Destination operand cannot be an immediate data. All the condition code flags are affected by this instruction.

Syntax: i. Sub mem/reg1, mem/reg2

[mem/reg1]←[mem/reg2]-[mem/reg2]

Ex: SUB BL,[SI]

SUB AX, BX

ii. SUB mem/data

[mem]←[mem]-data

Ex: SUB start, 02H

SUB [SI],0712H

iii. SUB A,data

[A]←[A]-data

Ex: SUB AL, 02H

SUB AX, 1211H

Examples with addressing modes:

1. SUB 0100H	Immediate [destination AX]
2. SUB AX, BX	Register
3. SUB AX,[5000H]	Direct
4. SUB [5000H], 0100	Immediate

SBB: Subtract with Borrow: The subtract with borrow instruction subtracts the source operand and the borrow flag (CF)which may reflect the result of the previous calculations, from the destination operand .Subtraction with borrow ,here means subtracting 1 from the subtraction obtained by SUB ,if carry (borrow) flag is set.

The result is stored in the destination operand. All the conditional code flags are affected by this instruction.

Syntax: i. SBB mem/reg1,mem/reg2

[mem/reg1] ← [mem/reg1]-[mem/reg2]-CY

Ex: SBB BL,[SI]

SBB AX,BX

ii. SBB mem,data

[mem] ← [mem]-data-CY

Ex: SBB Start,02H

SBB [SI],0712H

iii. SBB reg,data

[reg] ← [reg]-data-CY

Ex: SBB CL,05H

SBB DX,0132H

iv. SBB A,data
[A] ← [A]-data-CY

Ex: SBB AL,02H
SBB AX,1211H

INC: Increment: This instruction increments the contents of the specified register or memory location by 1. All the condition flags are affected except the carry flag CF. This instruction adds a to the content of the operand. Immediate data cannot be operand of this instruction.

Syntax: i. INC reg16
[reg 16]←[reg 16]+1
Ex: INC BX

ii. INC mem/reg 8
[mem]←[mem]+1
[reg 8]←[reg 8]+1

Ex: INC BL
INC SI

Segment register cannot be incremented. This operation does not affect the carry flag.

Examples with addressing modes:

1. INC AX	Register
2. INC [BX]	Register indirect
3. INC [5000H]	Direct

DEC: Decrement: The decrement instruction subtracts 1 from the contents of the specified register or memory location. All the condition code flags except carry flag are affected depending upon the result. Immediate data cannot be operand of the instruction.

Syntax: i. DEC reg16
[reg 16]←[reg 16]-1
Ex: DEC BX

ii. DEC mem/reg8
[mem]←[mem-1
[reg 8]←[reg 8]-1
Ex: DEC BL

Segment register cannot be decremented.

Examples with addressing mode:

1. DEC AX	Register
2. DEC [5000H]	Direct

MUL: Unsigned multiplication Byte or Word: This instruction multiplies unsigned byte or word by the content of AL. The unsigned byte or word may be in any one of the general-purpose register or memory locations. The most significant word of result is stored in DX, while the least significant word of the result is stored in AX. All the flags are modified depending upon the result. Immediate operand is not allowed in this instruction. If the most significant byte or word of the result is '0' IF and OF both will be set.

Syntax: MUL mem/reg

For 8X8
[AX]←[AL]*[mem8/reg8]
Ex: MUL BL
[AX]←[AL]*[BL]
For 16X16
[DX][AX]←[AX]*[mem16/reg16]
Ex: MUL BX
$$\underset{\text{higher 16-bit}}{[DX]}\;\underset{\text{lower 16-bit}}{[AX]} \leftarrow [AX]*[BX]$$

Ex:
1. MUL BH ; [AX]←[AL]*[BH]
2. MUL CX ; [DX][AX]←[AX*[CX]
3. MUL WORD PTR[SI];[DX][AX]←[AX]*[SI]

IMUL: Signed Multiplication: This instruction multiplies a signed byte in source operand by a signed byte in AL or signed word in source operand by signed word in AX. The source can be a general purpose register, memory operand, index register or base register, but it cannot be an immediate data. In case of 32-bit results, the higher order word (MSW) is stored in DX and the lower order word is stored in AX. The AF, PF, SF and ZF flags are undefined after IMUL. If AH and DH contain parts of 16 and 32-bit result respectively, CF and OF both will of set. The AL and AX are the implicit operands in case of 8-bit and 16-bit multiplications respectively. The unused higher bits of the result are filled by sign bit and CF, AF are cleared.

Syntax: IMUL mem/reg
For 8X8
[AX]←[AL]*[mem8/reg8]
Ex: IMUL BL
[AX]←[AL]*[BL]

For 16X16
[DX][AX]←[AX*[mem16/reg16]
Ex: IMUL BX
[DX][AX]←[AX]*[BX]

Memory or register can be 8-bit or 16-bit and this instruction will affect carry flag & overflow flag.

Ex: 1. IMUL BH
2. IMUL CX
3. IMUL [SI]

DIV: Unsigned division: This instruction performs unsigned division. It divides an unsigned word or double word by a 16-bit or 8-bit operand. The dividend must be in AX for 16-bit operation and divisor may be specified using any one of the addressing modes except immediate. The result will be in AL (quotient) while AH will contain the remainder. If the result is too big to fit in AL, type 0(divide by zero) interrupt is generated. In case of a double word dividend (32-bit), the higher word should be in DX and lower word should be in AX. The divisor may be specified as already explained. The quotient and the remainder, in this case, will be in AX and DX respectively. This instruction does not affect any flag.

Syntax: DIV mem/reg

Ex: DIV BL (i.e. [AX]/[BX])

For 16 ÷ 8 $\dfrac{[AX]}{[\text{mem 8/reg 8}]} \rightarrow$ [AH]← Remainder
[AL]← Quotient

For 32 ÷ 16 $\dfrac{[DX]\,[AX]}{[\text{mem } 16/\text{reg } 16]} \rightarrow \begin{array}{l}[DX] \leftarrow \text{Remainder} \\ [AX] \leftarrow \text{Quotient}\end{array}$

Ex: DIV BX (i.e. $\dfrac{[DX][AX]}{[BX]}$)

IDIV: Signed Division: This instruction performs same operation as the DIV instruction, but it with signed operands the results are stored similarly as in case of DIV instruction in both cases of word and double word divisions the results will also be signed numbers. The operands are also specified in the same way as DIV instruction. Divide by zero interrupt is generated, if the result is too big to fit in AX (16-bit dividend operation) or AX and DX (32-bit dividend operation) all the flags are undefined after IDIV instruction.

AAA: ASCII Adjust after addition: The AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. The AAA instruction converts the resulting contents of AL to unpacked decimal digits. After the addition, the AAA instruction examines the lower 4-bits of AL to check whether it contains a valid BCD number in the range 0 to 9. If it is between 0 to 9 and AF is zero, AAA sets the 4- higher order bits of AL to 0. The AH must be cleared before addition. If the lower digit of AL is between 0 to 9 and AF is set, 06 is added to AL. The upper 4-bits of AL are cleared and AH is incremented by one. If the value of lower nibble of AL is greater than 9 then the AL is incremented by 06, AH is incremented by 1, the AF and CF flags are set to 1, and the higher4-bits of AL are cleared to 0. The remaining flags are unaffected. The AH is modified as sum of previous contents (usually 00) and the carry from the adjustment, as shown in Fig1.7. This instruction does not give exact ASCII codes of the sum, but they can be obtained by adding 3030H to AX.

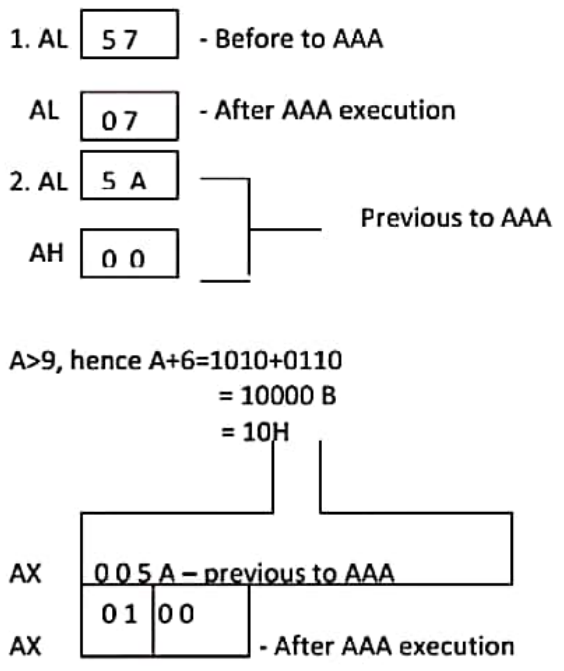

Fig1.7 ASCII Adjust After Addition Instruction

AAS: ASCII Adjust After Subtraction: AAS instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. If the lower 4-bits of AL register are greater than 9 or if the AF flag is one, the AL is decremented by 6 and AH is decremented by 1, the CF and AF are set to 1. Otherwise, the CF and AF are set to 0, the result needs to no correction. As a result, the upper nibble of AL is 00 and the lower nibble may be any number from 0 to 9. The procedure similar to the AAA instruction AH is modified as difference of previous contents (usually 0) of AH and the borrow for adjustment.

AAM: ASCII Adjust after Multiplication: This instruction, after execution, converts the product available in AL into unpacked BCD format. This follows a multiplication instruction. The lower byte of result (unpacked) remains in AL and the higher byte of result remains in AH.

The example given below explains execution of the instruction. Suppose, a product is available in AL, say AL=5D. AAM instruction will form unpacked BCD result in AX. DH is greater than 9, so add of 6(0110) to it D+6=13H. LSD of 13H is the lower unpacked byte for the result. Increment AH by 1, 5+1=6 will be the upper unpacked byte of the result. Thus after the execution, AH=06 and AL=03.

AAD: ASCII Adjust before Division: Though the names of these two instructions (AAM and AAD) appear to be similar, there is a lot of difference between their functions. The AAD instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing number the two unpacked BCD digits in AX by an unpacked BCD byte. PF, SF, ZF are modified while AF, CF, OF are undefined, after the execution of the instruction AAD. The example explains the execution of the instruction.

Let AX contain 0508 unpacked BCD for 58 decimal and DH contain 02H.
Ex:

AX	5	8

AAD result in AL	0	3A	58D=3AH in AL

The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH.

DAA: Decimal Adjust Accumulator: This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL. If the lower nibble is greater than 9, after addition or if AF is set, it will add 06 to the lower nibble in AL. After adding 06 in the lower nibble of AL, if the upper nibble of AL is greater than 9 or if carry flag is set, DAA instruction adds 60H to AL.

The example given below explains the instruction:

```
i. AL=53CL=29
        ADD AL, CL      ;       AL ←(AL) + (CL)
                        ;       AL←53+29
                        ;       AL←7C
                        ;       AL←7C+06(as C>9)
                        ;       AL←82
ii. AL=73       CL=29
        ADD AL,CL       ;       AL←AL+CL
                        ;       AL←73+29
                        ;       AL←9C
                        ;       AL←9C
        DAA             ;       AL←02 & CF=1
        AL=73
          +
        CL=29
        ________
          9C
          +6
        ________

         A2
          +60
        ________
        CF=1 02 in AL
```

The instruction DAA affects AF, CF, PF and ZF flags. The OF flag is undefined.