Chapter 2
Assembly Language Programming

Contents:

- Intro to assembly Language programming
- Addressing Modes
- Data Transfer, Arithmetic and Logical Instructions
- String Instructions - Machine Control Instructions

Introduction to assembly language

- There are three language levels that can be used to write a program
- **Machine Language:**
  - The binary form of the program is referred to as machine language because it is the form required by the machine.
- **Assembly Language** is a low-level language. Deals directly with the internal structure of CPU.
- **Assembly Languages** provided **mnemonics** for machine code instructions.
- **Mnemonics** refer to codes and abbreviations to represent instructions and make it easier for the users to remember.

- **Assembler** translates Assembly language program into machine code.
- In high-level languages, Pascal, Basic, C,C++,Java; the programmer does not have to be concerned with internal details of the CPU.
- **Compilers** translate the program into machine code.

✓ However, it is difficult, if not impossible, for a programmer to

memorize the thousands of binary instruction codes for a CPU such as the 8086.

Assignment of assembly language

- In Java, assignment takes the form:

```
x = 42 ;
y = 24;
```

- In assembly language we carry out the same operation but we use an instruction to denote the assignment operator (“=” in Java).

```
mov     x, 42
mov     y, 24
```

- The **mov** instruction carries out assignment in 8086 assembly language.
- It allows us to place a number in a register or in a memory location (a variable) i.e. it assigns a value to a register or variable.
- **Example: Store the ASCII code for the letter A in register bx.**
- A has ASCII code 65D (01000001B, 41H)
- The following **mov instruction carries out the task:**
- **mov bx, 65d**

- We could also write it as:

  mov bx, 41h

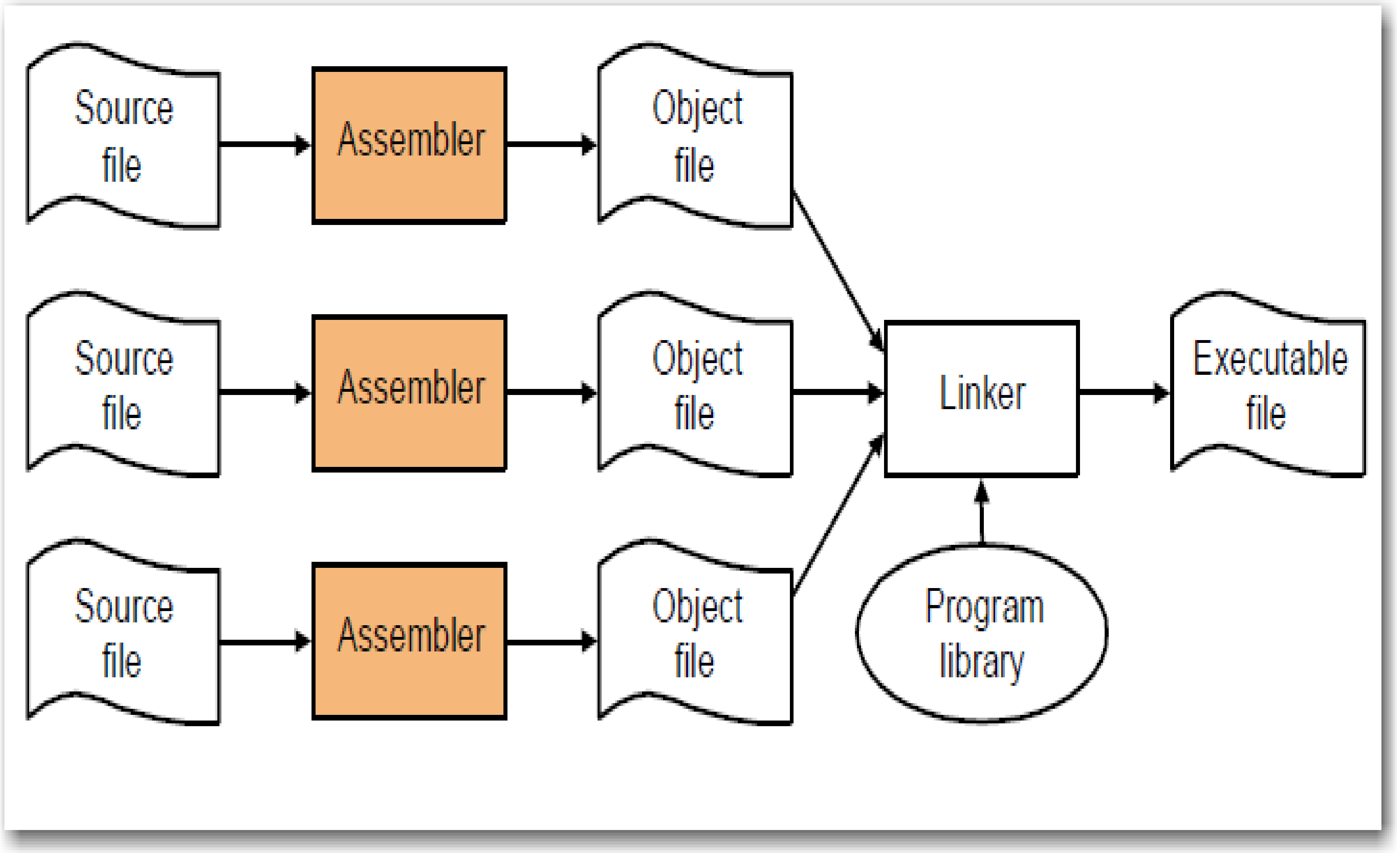
  or **mov bx, 01000001b**

  or **mov bx, ‘A’**
- All of the above are equivalent. They each carry out exactly the same task, namely the binary number representing the ASCII code of A is copied into the bx register.
- we could also have written it as:
- **mov bl, 65d**
- **mov bl, ‘A’**
- Since register bl represents the low-order byte of register bx.

- **Note:** The 8086 Assembler converts a **character constant i.e.** a character in single quotes (e.g. ‘A’) to its ASCII code automatically.
- This is a very useful feature and it means that you can specify many characters without having to look up their ASCII code. You simply enclose the character in single quotes.

Compilers, Assemblers, Linkers & Loaders

Compilation Process



Assembler and the Source Program

- ✓ Assembly language program (.asm) file—known as "source code" converted to machine code by a process called "assembling".
- ✓ Assembling performed by a software program — an "8088/8086 assembler".
  - – "Machine (object) code" that can be run on a PC is output in the executable (.exe) file.
- ✓ MASM—Microsoft 80x86 macroassembler allows a complete program to be assembled in one step

- ❖ After typing the program save the file with appropriate file name with an extension *.ASM*

  Ex: Add.ASM

❖ Assembling an ALP

- ➢ To assemble an ALP we need executable file called MASM.EXE. Only if this file is in current working directory we can assemble the program. The command is: *MASM<filename.ASM>*
- ❖ If the program is free from all syntactical errors, this command will give the **OBJECT file.** In case of errors it list out the number of errors, warnings and kind of error.
- ❖ **Note:** No object file is created until all errors are rectified.
- ❖ **Linking:**
- ➢ After successful assembling of the program we have to link it to get **Executable file.** The command is *LINK<File name.OBJ>*

ADDRESSING MODES OF 8086

- Addressing modes describe the types of operands and the way they are accessed for executing an instruction.
- There are **eight** addressing modes for 8086 instructions to specify operand.
  - Register addressing
  - **Immediate Addressing**
  - **Direct Addressing**
  - **Register Indirect Addressing**
  - **Based Addressing**
  - **Indexed addressing**
  - **Based Indexed Addressing**
  - **Based Indexed Addressing with displacement**

1. Register Addressing

- In this mode of addressing an 8-bit or 16 bit general purpose register contains an operand.

Example: **MOV BX,CX**; move the content of CX reg.to BX reg.

ADD AL,CH

ADD CX,DX

2. Immediate Addressing

- In this addressing mode, the operand is contained in the instruction itself.
  - **MOV AL,58H** **; move 58H to AL register.**
  - **MOV BX, 0354H** **; move 0354H to BX reg.**
  - **ADD AX, 0395H** **; Add 0395H to the content of AX reg.**

3. Direct Addressing

- This mode of addressing an effective address (or offset) is given in the instruction itself.
- **MOV AL, [0300H]** ; this instruction will move the content of the offset address 0300H to AL.
- **MOV [0401H], AX** ; this instruction will move the content of AX to the offset address 0401H.

4. Register Indirect Addressing

The operand's offset is in the base register, BX or base pointer BP or in an index register (SI or DI) specified in the instruction.

- **Example:** **ADD CX,[BX] ;** this will add the contents of the memory locations addressed by register BX to the register CX.

  MOV DX,[SI]; the content of the memory location addressed by SI will move to DX.

Offest

- An offset is called **effective address**
- The offset is determined by adding any combination of offset address elements Displacement, base and index.
  - The combination depends on the addressing mode of the instruction to be executed.
  - **Displacement:** It is an 8-bit or 16-bit immediate value given in the instruction.
  - **Base:** content of the base register, BX or BP.
  - **Index:** content of the index register, SI or DI.

5. Based Addressing

- The operand's offset is the sum of the contents of the base register , BX or BP and an 8-bit or 16-bit displacement.
- Offset (effective address)= [BX or BP +8-bit or 16-bit displacement].
- **ADD AL, [BX + 04];** case of 8-bit displacement. Suppose, BX contains 0301H. So 0301 + 04 =0305H & stored in AL
- **ADD AL, [BX + 1243H]; case of 16-bit displacement.**

6. Indexed Addressing

- the operand's offset is computed by adding an 8-bit or 16-bit displacement to the content of an index register SI or DI.
- Offset= [SI or DI + 8-bit or 16-bit displacement].
- **Example: ADD AX, [SI + 08]**

  MOV CX, [SI + 1523H].

7. Based Indexed Addressing

- The operand’s offset is computed by adding the contents of a base register to the contents of an index register.
- Off set = [BX or BP] + [SI or DI]. BX is used as a base register for stack segment.
- **Example:- MOV AX, [BX + SI]**

  ADD CX, [BX + SI]

8. Based Indexed with Displacement

- The operand's offset is computed by adding a base register's contents, an index register's contents and an 8-bit or 16-bit displacement.
- Offset = [BX or BP] + [SI or DI] + Displacement. BX is used with data segment, where as BP is used with stack segment.
- **Example:- MOV AX, [BX + SI + 05]**

  ADD CX, [BX + SI + 1212H]

INSTRUCTION SET OF 8086

- Data Transfer Instructions
- Arithmetic instructions
- Bit Manipulation Instructions
- String instructions
- Program execution transfer instructions
- Processor control instructions

Data Transfer Instructions

- This type of instructions is used to transfer data from source operand to destination operand.
- All the store, move, load, exchange, pop, push, input and output instructions belong to this category.

Byte Or Word Transfer Instructions (MOV-copy)

- MOV D,S ⟶ (S) => (D)

MOV CX, 037AH	Put the immediate number 037AH in CX
MOV BL, [437AH]	Copy byte in DS at offset 437AH to BL
MOV AX, BX	Copy contents of register BX to AX
MOV DL, [BX]	Copy byte from memory at [BX] to DL.
MOV DS, BX	Copy word from BX to DS register

MOV RESULTS[BP],AX Copy AX to two memory locations-AL to the first location, AH to the second. EA of the first memory location is the sum of the displacement represented by RESULTS and contents of BP. Physical address = EA + SS.

MOV CS:RESULTS[BP],AX Same as the above instruction, but

physical address = EA + CS because of the **segment override prefix CS**.

XCHG instructions

- XCHG- exchanges the contents of a Reg with the contents of any other Reg or Mem location.
- This instruction cannot exchange segment registers or memory–to- memory data.
- It can use any addressing modes except immediate addressing.

Examples

XCHG AX,DX Exchange word in AX with word in DX

XCHG BL,CH Exchange byte in BL with byte in CH

XCHG AL,PRICES [BX] Exchange byte in AL with byte in memory at EA = PRICES [BX] in DS

XLAT/XLATB-Translate a Byte in AL

- The XLATB instruction is used to translate a byte from one code to another code.
- The instruction replaces a byte in the AL register with a byte pointed to by BX in a lookup table in memory.
  - It is used in -ASCII to 7-segment display conversion.
  - ASCII-to-EBCDIC conversion.
- **AL** contains offset of the element to be accessed from the beginning of the lookup table.

(AL) ←((DSx10) + (BX) + (AL)) ; translate

XLAT/XLATB…

- Before the XLATB execution, the lookup table containing the values for the new code must be put in memory, and the offset of the starting address of the lookup table must be loaded in BX.
- The code byte to be translated is put in AL.
- To point to the desired byte in the lookup table, the XLATB instruction adds the byte in AL to the offset of the start of the table in BX.
- It then copies the byte from the address pointed to by (BX + AL) back into AL.

E.g. Assume (**DS**)=0300H, (**BX**)=0100H, (**AL**)=41H (ASCII code for character A),

EBCDIC code for A = **C1H**

MOV AX, **SEGTABLE**; address of the segment containing look-up table

MOV DS, AX ; is transferred to DS

MOV AL, CODE ; code of the pressed key (say A=41) as offset is transferred in AL

MOV BX, **OFFSET TABLE;** offset of the code lookup table in BX

XLAT(MOV AL,[AL][BX]) ; find the equivalent code and store in AL

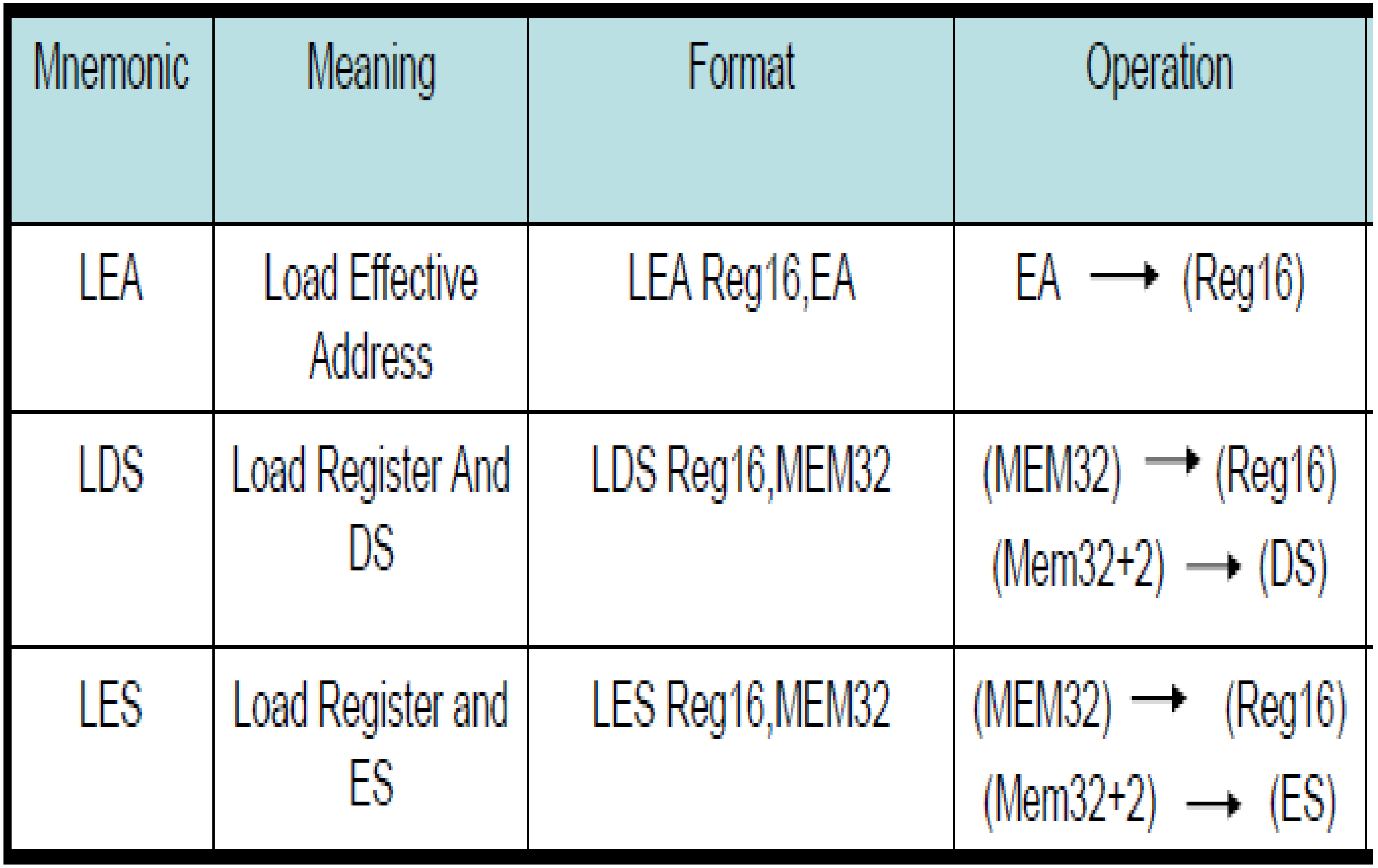
PA=DSx10 + (BX) + (AL) = 03000+0100H + 41H = 03141H

then (AL) ← (03141H)

i.e., (AL)= C1H

Note:-DS and BX must be initialized first before executing XLAT.

LEA, LDS, LES Instructions:



Mnemonic	Meaning	Format	Operation
LEA	Load Effective Address	LEA Reg16,EA	EA $\rightarrow$ (Reg16)
LDS	Load Register And DS	LDS Reg16,MEM32	(MEM32) $\rightarrow$ (Reg16) (Mem32+2) $\rightarrow$ (DS)
LES	Load Register and ES	LES Reg16,MEM32	(MEM32) $\rightarrow$ (Reg16) (Mem32+2) $\rightarrow$ (ES)

- Examples

LDS BX, [4326] Copy contents of memory at displacement 4326H in DS to BL, contents of 4327H to BH. Copy contents at displacement of 4328H and 4329H in DS to DS register.

LDS SI, STRING_POINTER

Copy contents of memory at displacements STRING_POINTER and STRING_POINTER+1 in DS to SI register. Copy contents of memory at displacements STRING_POINTER+2 and STRING POINTER+3 In DS to DS register. DS:SI now points at start of desired string.

LES BX, [789AH] Contents of memory at displacements 789AH and 789BH In DS copied to BX. Contents of memory at displacements 789CH and 789DH in DS copied to ES register.

LES DI, [BX] Copy contents of memory at offset [BX] and offset [BX+1] in DS to DI register. Copy contents of memory at offsets [BX + 2] and [BX + 3] to ES register.

Input-output instructions

- **IN /OUT Ins:**. -IN -input from the port.

  -OUT-Output to the port.
- **AL** and **AX** are the allowed destinations for 8-bit and 16-bit I/O operations.
- **DX** is the implicit address of I/O ports.
- **IN AL, 30H ; read from an 8-bit port whose address is 0030H in to AL**
- **IN AX, 0400H;**
- **IN AX ; read from a 16-bit port whose implicit address is in DX in to AX**
- **IN AX, DX**
- **OUT 30H;**
- **OUT 0400H**
- **OUT AX ;**
- **OUT AX, DX**

PUSH/POP instructions

- **PUSH-** push to stack. Pushes the contents of the specified Reg/Mem location on to stack.
- Higher byte is pushed first, then lower byte.
- **PUSH Reg16**
- **PUSH Mem16**
- **PUSH Seg ; Seg can be any of CS, DS, ES, SS**
- **PUSHA ; save all 16-bit registers except Seg registers**
- **PUSHF ; save flags**
- **POP:.** performs the inverse operation of a PUSH instruction.
- **POP Seg ; Seg can be any of DS, ES, SS, but not CS**
- **Eg**
- **POP CS** ; invalid, **CS** cannot popped
- **POP [5000H]**

Arithmetic Instructions

Addition Instructions:

○**ADD** Add specified byte-to-byte or specified word to word.

Format:

operand1 = operand1 + operand2

```
MOV AL, 5        ; AL = 5
ADD AL, -3       ; AL = 2
RET
```

○**ADC-Add with Carry-ADC**

Format:

operand1 = operand1 + operand2 + CF

Example:

```
STC              ; set CF = 1
MOV AL, 5        ; AL = 5
ADC AL, 1        ; AL = 7
RET
```

○INC-increment- INC Destination

- **Format:**
- operand = operand + 1
- **Example:**

```
        MOV AL,   4
        INC AL      ; AL = 5
```

○ AAA-ASCII Adjust After Addition

- This allows us to add the ASCII codes for two decimal digits.
- After the addition, the AAA Instruction is used to make sure the result is the correct BCD
- **Example:**

```
  Assume AL = 0 0 1 1 0 1 0 1, ASCII 5
         BL = 0 0 1 1 1 0 0 1, ASCII 9
ADD AL,BL      Result:  AL= 0 1 1 0 1 1 1 0 = 6EH, which is
                                              incorrect BCD
AAA                     14 decimal
```

DAA-Decimal Adjust after BCD Addition

- Decimal adjust After Addition.
- Corrects the result of addition of two BCD values.

```
        AL = 0101 1001 = 59 BCD
        BL = 0011 0101 = 35 BCD
  ADD AL, BL              AL = 1000 1110 = 8EH
  DAA            Add 01 10 because 1110 > 9 AL = 1001 0100 = 94 BCD
```

Subtraction Instructions:

- **SUB-Subtract**
- **Format:**
- operand1 = operand1 - operand2
- **Example:**

```
MOV AL, 5
SUB AL, 1          ; AL = 4
RET
```

- **SBB-Subtract with Carry**
- **Format:**
- **operand1 = operand1 - operand2 – CF**
- **Example:**

```
STC
MOV AL, 5
SBB AL, 3          ; AL = 5 - 3 - 1 = 1
RET
```

DEC-Decrement Destination Register or Memory

- This Instruction subtracts 1 from the destination.
- Format :
- operand = operand – 1

Example1 : DEC CL

DEC BP

Example2:

```
MOV AL, 255          ; AL = 255
DEC AL               ; AL =  254
```

NEG- (Form 2's Complement)

Format: Invert all bits of the operand.

Add 1 to inverted operand.

Example:

NEG AL

NEG BX

- **CMP-Compare Byte or Word**
  The comparison is done by subtracting the source byte or word from the destination byte or word.
- **Format:** operand1 - operand2

Example:

```
 MOV AL, 5
 MOV BL, 5
CMP  AL, BL     ; ZF = 1
```

Multiplication Instruction

- **MUL---Multiply Unsigned Bytes or Words**

- when operand is a **byte:**
- AX = AL * operand.
- when operand is a **word:**
  - (DX AX) = AX * operand.

Example: MUL BH	AL times BH, result in AX
MUL CX	AX times CX, result high word in DX, low word in AX

- **IMUL-Multiply Signed Numbers**
  This instruction multiplies a signed byte from some source times a signed byte in AL or a signed word from some source times a signed word in AX.

Example: IMUL BH	Signed byte in AL times signed byte in BH, result in AX
IMUL AX	AX times AX, result in DX and AX

Division Instruction

- **DIV-Unsigned Divide**
- This instruction is used to divide an **unsigned word by a byte** or to divide an unsigned **double word (32 bits) by a word**.
- **Example:**

```
MOV AX, 203                      ; AX = 00CBh
MOV BL, 4
DIV BL                           ; AL = 50 (32h), AH = 3
RET
```

IDIV-Divide by Signed Byte or Word

- This instruction is used to divide a signed word by a signed byte, or to divide a signed double word (32 bits) by a signed word.
- **Example:**

```
MOV AX, -405
MOV BL, 4
IDIV BL ; AL = -101 , AH = -1
 RET
```

3. Bit Manipulation Instructions:

Logical Instructions:

- **NOT- Invert each bit of the operand.**
- **Format:**
- **If bit is 1 turn it to 0.**
- **If bit is 0 turn it to 1.**
- **Example:**

```
MOV AL, 00011011b
NOT AL ; AL = 11100100b
RET
```

- **AND-Logical AND between all bits of two operands.**
- These rules apply:
- 1 AND 1 = 1
- 1 AND 0 = 0
- 0 AND 1 = 0
- 0 AND 0 = 0

- **Example:**

```
MOV AL, 'a'              ; AL = 01100001b
AND AL, 11011111b        ; AL = 01000001b ('A')
RET
```

OR- OR Logical between all bits of two operands.

- These rules apply:
- 1 OR 1 = 1
- 1 OR 0 = 1
- 0 OR 1 = 1
- 0 OR 0 = 0
- **Example:**

```
MOV AL, 'A'              ; AL = 01000001b
OR AL, 00100000b         ; AL = 01100001b ('a')
RET
```

○ Shift Instructions:

- **SHL-Shift Operand Bits Left, Put Zero in LSB(s)**
- **CF ← MSB ← LSB ← 0**

Example: MOV AL, 11100000b

```
SHL AL, 1        ; AL = 11000000b, CF=1.
RET
```

- **SHR-Shift Operand Bits Right, Put Zero in MSB(s)**

0 → MSB → LSB → CF

CF contains the bit most recently shifted in from the LSB.

Example1:

```
MOV AL, 00000111b
SHR AL, 1 ; AL = 00000011b, CF=1.
RET
```

- **ROL-Rotate Bits of Operand Left, MSB to LSB**

Example1: ROL AX, 1   Word in AX 1bit position left, MSB to LSB and CF

- **Example2 :**

```
MOV AL, 1Ch           ; AL = 00011100b
ROL AL, 1              ; AL = 00111000b, CF=0.
RET
```

4. String Instructions

- **MOVS/MOVSB/MOVSW -Move String Byte or String Word**
  This instruction copies a byte or a word from a location in the data segment to a location in the extra segment.

- The offset of the source byte or word in the data segment must be in the SI register.
- The offset of the destination in the extra segment must be contained in the DI register.
- For multiple-byte or multiple-word moves, the number of elements to be moved is put in the CX register so that it can function as a counter.
- After the byte or word is moved, SI and DI are automatically adjusted to point to the next source and the next destination.

Example

MOV SI, OFFSET SOURCE_STRING Load offset of start of source string in DS into SI

MOV DI, OFFSET DESTINATION-STRING   Load offset of start of destination string in ES into DI

CLD   Clear direction flag to auto increment SI & DI after move

MOV CX, 04H   Load length of string into CX as counter

REP MOVSB   Decrement CX and copy string bytes until CX = 0

After the move, SI will be 1 greater than the offset of the last byte in the source string. DI will be 1 greater than the offset of the last byte in the destination string.

CX will be 0.

- **LODS/LODSB/LODSW Load String Byte into AL or Load String Word into AX**
- This instruction copies a byte from a string location pointed to by SI to AL, or a word from a string location pointed to by SI to AX.

EXAMPLE:

CLD ;Clear direction flag so SI is auto incremented.

MOV SI, 0301H ;memory address in SI.

LODSB ; Load AL with the content of memory locations specified by SI register.

○STOS/STOSB/STOSW-Store Byte or Word in String

- The STOS instruction copies a byte from AL or a word from AX to a memory location in the extra segment pointed to by DI.
- In effect, it replaces a string element with a byte from AL or a word from AX.
- After the copy, DI is automatically incremented or decremented to point to the next string element in memory.
- If the direction flag (DF) is cleared, then DI will automatically be incremented by 1 for a byte string or incremented by 2 for a word string.
- If the direction flag is set, DI will be automatically decremented by 1 for a byte string or decremented by 2 for a word string.

❖ **Store AL or AX in to the memory locations addressed by DI.**

❖ **Example:**

```
 MOV AX,   7642H                        ;Data in AX
MOV DI, 0302H                           ; memory address in DI.
STOSW                                   ; store [AX] in [DI].
```

Result

```
0302     42H
0303     76H
```

oREP/REPE/REPZ/REPNE/REPNZ- (Prefix) Repeat String Instruction until Specified Conditions Exist

- REP is a prefix, which is written before one of the string instructions. It will cause the CX register to be decremented and the string instruction to be repeated until CX = 0.
- REPE and REPZ are two mnemonics for the same prefix.
- They stand for Repeat if Equal and Repeat if Zero, respectively.
- REPE or REPZ is often used with the Compare String instruction or with the Scan String instruction.
- REPE or REPZ will cause the string instruction to be repeated as long as the compared bytes or words are equal (ZF = 1) and CX is not yet counted down to zero.
- In other words, there are two conditions that will stop the repetition: CX = 0 or string bytes or words not equal.
- RFPNE or REPNZ will cause the string instruction to be repeated until the compared bytes or words are equal (ZF = 1) or until CX = 0 (end of string).

Program Execution Transfer Instructions

- **Conditional Transfer Instructions**
- **Iteration Control Instructions**
- **Interrupt Instructions**

Conditional Transfer Instructions:

JA/JNBE	Jump if above/jump if not below or equal.
JAE/JNB	Jump if above or equal/jump if not below.
JB/JNAE	Jump if below/jump if not above or equal.
JBE/JNA	Jump if below or equal/jump if not above.
JC	Jump if carry flag CF = 1.
JE/JZ	Jump if equal/jump if zero flag ZF = 1.
JG/JNLE	Jump if greater/jump if not less than or equal.
JGE/JNL	Jump if greater than or equal/ Jump if not less than.
JL/JNGE	Jump if less than/jump if not greater than or-equal.
JLE/JNG	Jump if less than or equal/jump if not greater than.
JNC	Jump if no carry (CF = 0).
JNE/JNZ	Jump if not equal/jump if not zero (ZF = 0).
JNO	Jump if no overflow (overflow flag OF = 0).
JNP/JPO	Jump if not parity/jump if parity odd (PF = 0).
JNS	Jump if not sign (sign flag SF= 0).
JO	Jump if overflow flag OF = 1.
JP/JPE	Jump if parity/jump if parity even (PF = 1).
JS	Jump if sign (SF=1)

- ## Iteration Control Instructions:

LOOP	Loop through a sequence of instructions until CX = 0
LOOPE/LOOPZ	Loop through a sequence of instructions while ZF=1 and CX ≠0
LOOPNE/LOOPNZ	Loop through a sequence of instructions while ZF = 0 and CX ≠ 0.
JCXZ	Jump to specified address if CX = 0.

- ## Interrupt Instructions:

INT	Interrupt program execution, call service procedure (ISR- Interrupt Service Routine).
INTO	Interrupt program execution if OF = 1.
IRET	Return from interrupt service procedure to main program.

Process Control Instructions

Flag Set/Clear Instructions:

STC	Set carry flag CF to 1.
CLC	Clear carry flag CF to 0.
CMC	Complement the state of the carry flag CF.
STD	Set direction flag DF to 1.
CLD	Clear direction flag DF to 0.
STI	Set interrupt enable flag to 1 (enable INTR input).
CLI	Clear interrupt enable flag to 0 (disable INTR input).

○ External Hardware Synchronization Instructions:

HLT	Halt (do nothing) until interrupt or reset.
WAIT	Wait (do nothing) until signal on the TEST pin is low.
ESC	Escape to external coprocessor such as 8087 or 8089
LOCK	An instruction prefix. Prevents another processor from taking the bus (in MAX mode)

○ No Operation Instruction:

NOP	No action except fetch and decode.