5.2 Macros

- Macro is a group of instructions. The macro assembler generates the code in the program each time where the macro is 'called'. Macros can be defined by MACRO and ENDM assembler directives. Creating macro is very similar to creating a new opcode that can be used in the program, as shown below.

Example : Macro definition for initialization of segment registers.

```
INIT MACRO        ; Define macro
MOV AX,@data      ;
MOV DS            ;  } Body of macro definition
MOV ES,AX         ;
ENDM        ; End macro
```

- It is important to note that macro sequences execute faster than procedures because there are no CALL and RET instructions to execute. The assembler places the macro instructions in the program each time when it is invoked. This procedure is known as **Macro expansion**.

5.2.1 Comparison of Procedure and Macro

Sr. No.	Procedure	Macro
1.	Accessed by CALL and RET instruction during program execution.	Accessed during assembly with name given to macro when defined.
2.	Machine code for instructions is put only once in the memory.	Machine code is generated for instructions each time when macro is called.
3.	With procedures less memory is required.	With macros more memory is required.
4.	Parameters can be passed in registers, memory locations, or stack.	Parameters passed as part of statement which calls macro.

Ex. 3.2.8 : *Write assembly language instruction of 8086 microprocessor to :*

i) Copy 1000 H to register BX.

ii) Rotate register BL left four times.

MSBTE : Summer-16, 17, Marks 3

Sol. :

i) MOV BX, 1000H

ii) MOV CL, 04H

ROL BL, CL

Ex. 3.2.9 : *What will be the content of register AL after the execution of last instruction ?*

MOV AL 02H

MOV BL, 02H

SUB AL, BL

MUL 08H

MSBTE : Summer-17, Marks 4

Sol. :

MOV AL, 02H → AL = 02H

MOV BL, 02H → AL = 02H

SUB AL, BL → AL = 00H

Ex. 3.2.9 : *Write an appropriate 8086 instruction to perform following operation :*

i) Initialize stack of 4200H.

ii) Multiply AL by 05H.

MSBTE : Summer-[illegible], Marks 4

Sol. :

i) MOV SP, 4200H

ii) MOV AL, 05H

MUL AL

... language instructions of 8086

MOVS / MOVSB / MOVSW

- These instructions copy a byte or 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. If the direction flag is 0, then SI and DI will be incremented by 1 after a byte move and they will incremented by 2 after a word move. If the DF is a 1, then SI and DI will be decremented by 1 after a byte move and they will be decremented by 2 after a word move. MOVS affects no flags.
- The way to tell the assembler whether to code the instruction for a byte or word move is to add a "B" or a "W" to the MOVS mnemonic. MOVSB, for example, says move a string as bytes. MOVSW says move a string as words.

Examples :

```
CLD                 ; Clear Direction Flag to autoincrement SI and DI
MOV AX, 0000H
MOV DS, AX          ; Initialize data segment register to 0
MOV ES, AX          ; Initialize extra segment register to 0
MOV SI, 2000H       ; Load offset of start of source string into SI
MOV DI, 2400H       ; Load offset of start of destination into DI
MOV CX, 04H         ; Load length of string in CX as counter
REP MOVSB           ; Decrement CX and MOVSB until CX will be 0.
```

Transducers:- A transducer is a device that converts energy from one form to another form. (energy).

Passive transducers:- Thermocouple, photo diode

1.4 Register Organization (8086 Programming Model)

MSBTE : Winter-15,16,17, Summer-15,16,17,18

- The 8086 has a powerful set of registers.
- It includes general purpose registers, segment registers, pointers and index registers, and flag register.

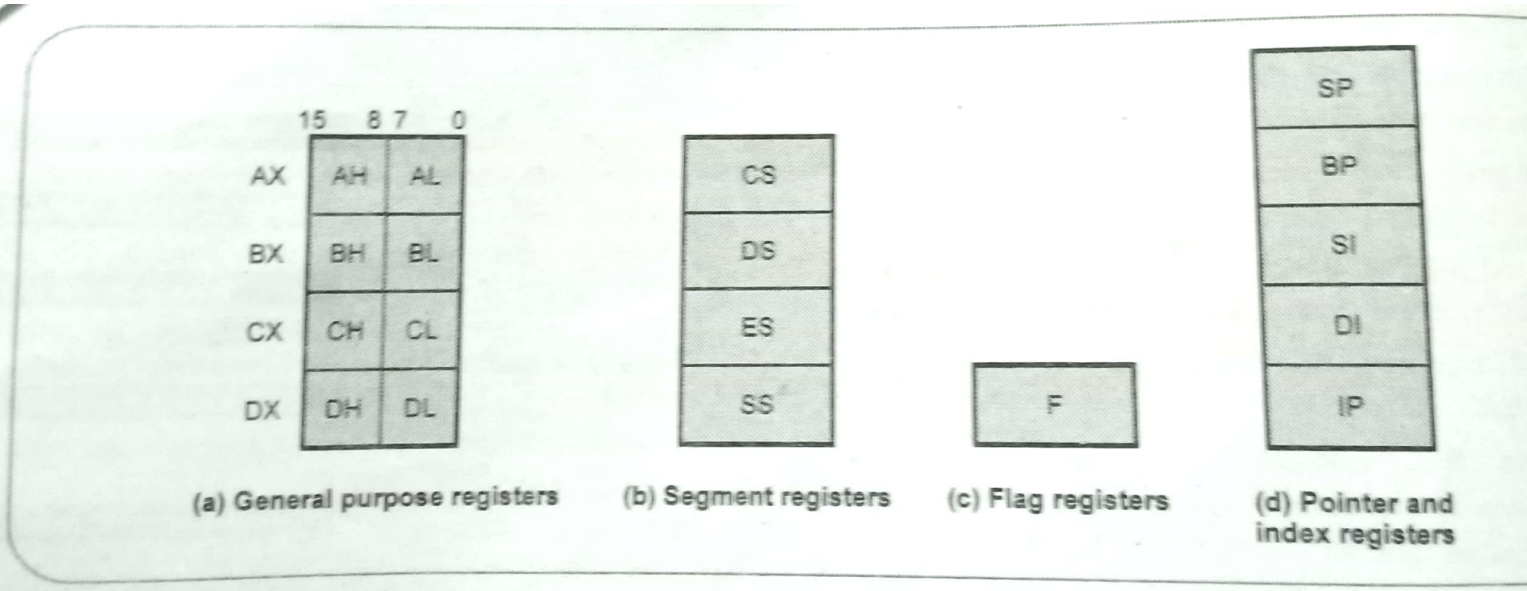

Fig. 1.4.1 Register organization of 8086

- The Fig. 1.4.1 shows the register organization of 8086. It is also known as **programmer's model of 8086.**
- The registers shown in programmer's model are accessible to programmer.
- As shown in the Fig. 1.4.1, all the registers of 8086 are 16-bit registers.
- The DX register is concatenated with AX to form 32-bit register for some MUL operations. It is also used to specify port a IN and OUT operations.

1.4.2 Segment Registers

- The physical address of the 8086 is 20-bi

Effective Address (EA). Note that EA is displacement of the desired location from the segment base. As mentioned before, the BIU generates a 20-bit physical address after shifting the contents of the desired segment register four bits to the left and then adding the 16-bit EA to it.

- There are six ways to specify Effective Address (EA) in the instruction.

a. Direct addressing mode
b. Register indirect addressing mode
c. Based addressing mode
d. Indexed addressing mode
e. Based indexed addressing mode
f. String addressing mode

1. Direct Addressing Mode : In this mode, the 16-bit Effective Address (EA) is taken directly from the displacement field of the instruction. The displacement (unsigned 16-bit or sign-extended 8-bit number) is stored in the location following the instruction opcode.

Example : MOV AL, [3000H] : This instruction will copy the contents of the memory location, at displacement of 3000H from the data segment base, into the AL register. Here, 3000H is the Effective Address (EA) which is written directly in the instruction.
Physical address : [DS] × [10H] + EA = [DS] × [10H] + 3000H

2. Register Indirect Addressing Mode : In this mode, the EA is specified in either a pointer register or an index register. The pointer register can be either base register BX or base pointer register BP and index register can be either Source Index (SI) register or Destination Index (DI) register. The 20-bit physical address is computed using DS and EA.

Example :

i. MOV [DI], BX; The instruction copies the 16-bit contents of BX into a memory location offset by the value of EA specified in DI from the current contents in DS. Now, if [DS] = 7205H, [DI] = 0030H, and [BX] = 8765H, then after MOV [DI], BX, content of BX (8765H) is copied to memory locations 72080H and 72081H.
Physical address : [DS] × [10H] + EA = [DS] × [10H] + [DI]

ii. MOV DL, [BP] ; This instruction copies the 8-bit ; contents in DL from the memory location offset by the value of EA specified in BP from the contents of SS. Because data addressed by BP are by default located in Stack Segment (SS).
Physical address : [SS] × [10H] + EA = [SS] × [10H] + [BP]

3. Base-Plus-Index-Addressing : Base-plus-index addressing is similar to indirect addressing because it indirectly addresses memory data. This addressing uses one base register (BP or BX) and one index register (DI or SI) to indirectly address memory. The base register often holds the beginning location of a memory array, while the index register holds the relative position of an element in the array. Remember that whenever BP addresses the memory data, the contents of stack segment, BP and index register are used to generate physical address.

Example : MOV CX, [BX + DI] ; In this mode, the EA is given by the sum of base register and index register, i.e. [BX] + [DI]
Physical address : [DS] × [10H] + EA = [DS] × [10H] + [BX] + [DI]

4. Register Relative Addressing : Register relative addressing is similar to base-plus-index addressing. Here, the data in a segment of memory are addressed by adding the displacement to the contents of a base or an index register (BP, BX, DI or SI). Remember that displacement should be added to the register within the [ ]. Displacement can be any 8-bit or 16-bit number.

Example : MOV CX [BX + 0003H] ; In this mode EA is given by sum of base register and displacement.

Note :

- Displacement can be subtracted from the register : MOV AL, [DI-2].
- Displacement can be an offset address appended to the front of the [ ] : MOV AL, OFF_ADD [DI + 4].

5. Base Relative Plus Index Addressing : The base relative plus index addressing mode is similar to the base plus index addressing mode, but it adds a displacement, besides using a base register and an index register to generate a physical address of the memory. This addressing mode is suitable to address data within the two dimensional array.

Example : MOV AL, [BX + SI + 10H] ; In this mode, the EA is the sum of base register, index register and displacement.

6. String Addressing Mode : This mode uses index registers. The string instructions automatically assume SI to point to the first byte or word of the source operand and DI to point to the first byte or word of the destination operand. The contents of SI and DI are automatically incremented (by clearing DF to 0 by CLD instruction) or decremented (by setting DF to 1 by STD instruction) to point to the next byte or word. The segment register for the source is DS. The segment register for the destination must be ES.

Example : MOVS BYTE ; If [DF] = 0, [DS] = 3000H, [SI] = 0600H, [ES] = 5000H, [DI] = 0400H, [30600H] = 38H, and [50400H] = 45H, then after execution of the MOVS BYTE, [50400H] = 38H,; [SI] = 0601H, and [DI] = 0401H.

3.1.1.3 Addressing Modes for Accessing I/O Ports (I/O Modes)

Standard I/O devices uses port addressing modes. For memory-mapped I/O, memory addressing modes are used. There are two types of port addressing modes : **direct** and **indirect**.

Direct port mode : Here, the port number is an 8-bit immediate operand. This allows fixed access to ports numbered 0 to 255.

Example :

```
OUT 05H, AL     ;  Sends the contents of AL to 8-bit port 05H.
IN AX, 80H      ;  Copies 16-bit contents of port 80H
```

Indirect port mode : Here the port number is taken from DX allowing 64 K 8-bit ports or 32 K 16-bit ports.

Example :

```
IN AL, DX       ;  if [DX] = 7890H, then it copies 8-bit content of port 7890H into AL.
IN AX, DX       ;  copies the 8-bit contents of ports 7890H and 7891H into AL
                ;  and AH, respectively.
```

Example : ASSUME CS : code, DS : Data, SS : stack.

.CODE : This directive provides shortcut in definition of the code segment. General format for this directive is as shown below.

.code [name]

The name is optional. It is basically specified to distinguish different code segments when there are multiple code segments in the program.

.DATA : This directive provides shortcut in definition of the data segment.

DB, DW, DD, DQ and DT : These directives are used to define different types of variables, or to set aside one or more storage locations of corresponding data type in memory. Their definitions are as follows :

DB - Define Byte
DW - Define Word
DD - Define Doubleword
DQ - Define Quadword
DT - Define Ten bytes

Examples :

```
AMOUNT DB 10H, 20H, 30H, 40H      ; Declare array of 4 bytes named AMOUNT
MES DB 'WELCOME'                  ; Declare array of 7 bytes and initialize with ASCII codes
                                  ; for letters in WELCOME.
```

DUP : The DUP directive can be used to initialize several locations and to assign values to these locations.

Format : Name Data_Type Num DUP (value)

```
Example :    TABLE DW 10 DUP (0)    ; Reserve an array of 10 words of memory and initialize
                                    ; all 10 words with 0. Array is named TABLE.
```

END : The END directive is put after the last statement of a program to tell the assembler that this is the end of the program module. The assembler ignores any statement after an END directive.

EQU : The EQU directive is used to redefine a data name or variable with another data name, variable or immediate value. The directive should be defined in a program before it is referenced.

Formats :

```
Numeric Equate      : name EQU expression
String Equate       : name EQU <string>
Example :           : Two EQU        ; Numeric value
NUM EQU <'Enter the first number :'>
MES DB NUM                           ; Replace with string
```

EVEN : EVEN tells the assembler to advance its location counter if necessary so that the next defined data item or label is aligned on an even storage boundary. This feature makes processing more efficient on processors that access 16 or 32 bits at a time.

Example :

```
EVEN LOOKUP DW 10 DUP (0)          ; Declares the array of ten words starting from even address
```

linked together to form the complete program. In order for the modules to link together correctly, any variable name or label referred to in other modules must be declared public in the module where it is defined. The PUBLIC directive is used to tell the assembler that a specified name or label will be accessed from other modules.

Format : PUBLIC Symbol [. . . .]

Example : PUBLIC SETPT ; Makes SETPT available for other modules.

SEGMENT and ENDS : An assembly program in .EXE format consists of one or more segments. The start of these segments are defined by SEGMENT directive and the ENDS statement indicates the end of the segment.

Format :
```
name    SEGMENT [options] ; Begin segment
.
.
name    ENDS                ; End segment
```

Example :
```
CODE    SEGMENT
.
.
.
CODE    ENDS
```

SHORT : A short is a operator. It is used to tell the assembler that only 1-byte displacement is needed to code a jump instruction. If the jump destination is after the jump instruction in the program, the assembler will automatically reserve 2-bytes for the displacement. Using the short operator saves 1-byte of memory by telling the assembler that it only needs to reserve 1-byte for this particular jump. The short operator should be used only when the destination is in the range of –128 bytes to +127 bytes from the address of the instructions after the jump.

Example : JMP SHORT NEAR_LABEL

.STACK : This directive provides shortcut in definition of the stack segment. General format for this directive is as shown below.

.stack [size]

The default size is 1024 bytes.

Example : .STACK 100 ;This reserves 100 bytes for the stack operation.

When stack is not use in the program .stack command can be omitted. This will reserve in the warning message "no stack segment" after linking the program. This warning may be ignored.

TITLE : The TITLE directive help to control the format of a listing of an assembled program. TITLE directive causes a title for a program to print on line 2 of each page of the program listing. Maximum 60 characters are allowed as title.

Format : TITLE text

Example : TITLE Program to find maximum number.

TYPE : It is an operator which tells assembler to determine the type of specified variable. Assembler determines the type of specified variable in number of bytes. For byte type variable the assembler gives a value of 1. For word type variable the assembler gives a value of 2 and for double word type variable the assembler gives a value of 4.

13. *State function of following assembly programming tool.*
i) Assembler ii) Linker

MSBTE : Summer-18, Marks 4

2.3 Assembler Directives

MSBTE : Summer-15, 16, 17, 18, Winter-15, 16, 17

There are some instructions in the assembly language program which are not a part of processor instruction set. These instructions are instructions to the assembler, linker and loader. These are referred to as **pseudo-operations** or as **assembler directives.** The assembler directives enable we to control the way in which a program assembles and lists. They act during the assembly of a program and do not generate any executable machine code.

There are many specialized assembler directives. Let us see the commonly used assembler directive in 8086 assembly language programming.

ALIGN : The align directive forces the assembler to align the next segment at an address divisible by specified divisor. The general format for this directive is as shown below.

ALIGN number

where number can be 2, 4, 8 or 16.

Example :
```
ALIGN 8   ; This forces the assembler to
          ; align the next segment at an
          ; address that is divisible by 8.
          ; The assembler fills the
          ; unused bytes with 0 for data
          ; and NOP instructions for
          ; code.
```

Usually ALIGN 2 directive is used to start the data segment on a word boundary and ALIGN 4 directive is used to start the data segment on a double word boundary.

ASSUME : The 8086, at any time, can directly address four physical segments which include a code segment, a data segment, a stack segment and an extra segment. The 8086 may contain a number of logical segments. The ASSUME directive assigns a logical segment to a physical segment at any given time. That is, the ASSUME directive tells the assembler what addresses will be in the segment registers at execution time.