8086 PROGRAMMING.

K. Priyanka

A machine language instruction format has one or more number of fields associated with it. The first field is called as operation code field or opcode field, which indicates the type of the operation to be performed by the CPU. The instruction format also contains other fields known as operand fields. The CPU executes the instruction using the information which reside in these fields.

Assembler directives:-

→ Assembly language programming is simple as compared to the machine language programming. The instruction mnemonics are directly written in the assembly language programs.

→ An Assembler is a program used to convert an assembly language program into the equivalent machine code modules which may further be converted to executable codes.

→ While doing these things, the assembler may find out syntax errors. The logical errors are not found out by the assembler.

→ For completing all the tasks, an assembler need some hints from the programmer. These types of hints are given to the assembler using some predefined alphabetical string called assemble directives

→ DB: Define byte:- This directive is used to reserve byte or byte of memory locations in the main memory.

EX:- KING DB 01H.02H,03H.

→ DW: Define word :- The DW directives serves the same purpose as the DB directive, but it now makes the assembler reserve the number of memory words (16-bit) instead of bytes.

EX:- KING DW 1234H,4567H,

→ DQ:- Define quad :- This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialise it with the specified values.

→DD:- Define Double :- This directive is used to direct the assembler to reserve 4 bytes of memory.

→DT:- Define Ten Bytes :- The DT directive directs the assembler to define the specified variable requiring 10-bytes for its storage and initialise the 10-bytes with the specified values.

→Assume : Assume logical segment Name :- The Assume directive is used to inform the assemble, the names of the logicals segments to be assumed for different segments used in the program.

EX:- assume CS:Code, DS:Data.

It should be placed at the start of each assembly language program.

→END: End of program :- The End directive marks the end of an assembly language program. Hence, it should be ensured that the End statement should be the last statement in the file and should not appear in between.

→ENDP : End of procedure :- In assembly language programming, the subroutines are called procedures. They may be independent program modules which returns particular results or values to the calling programs.

```
EX:-  Procedure star
        :
      Star ENDP
```

→ENDS : End of Segment :- This directive marks the end of a logical segment. The logical segments are assigned with the names using the Assume directive.

```
EX:-    Data   segments
          :
        Data Ends.
Assume CS:code, DS:data.
Code   Segment
         :
Code   Ends
End.
```

→ Even :- Align on Even memory Address :-

→ This directive updates the location counter to the next even address.

→ If the current location address is odd, it will be updated to next even address.

```
        EVEN
  procedure  Root
             :
             :
   Root      ENDP
```

→ Equ : Equate :- The directive EQU is used to assign a label with a value or a symbol.

EX1 :- LABEL EQU 0500H

EX2 :- ADDITION EQU ADD.

→ Group :- This directive is used to form logical Groups of segments with similar purpose or type. This directive is used to inform the assembler to form a logical group of the following segment Names. (Group of the Related segments).

program Group code, data, stack.

→ Label :- The label directive is used to assign a name to the current content of the location counter.

```
BACK :- mov ax, data
        :
        :
        JNZ Back
```

→ Name :- logical name of a module :- The Name directive is used to assign a name to an assembly language program module.

→ OFFSET :- offset of a Label :- when the assembler comes across the OFFSET operator along with a label,

EX:- Code segment

MOV SI, OFFSET List

Code Ends

ORG:- Origin:- The ORG directive directs the assembler to start the memory allotment for the particular segment from the declared address. in the ORG statement.

ORG 5000H

The program execution will start from location 5000H.

LENGTH:- Byte length of a label:- This directive is not available in MASM - This is used to refer to the length of a data array or a string.

MOV CX, length Array.

Instruction set of 8086/8088:-

The 8086/8088 instructions are categorised into the following main types.

1) Data copy / Transfer instructions
2) Arithmetic and logical instructions
3) Branch instructions.
4) Loop instructions.
5) Machine control instructions.
6) Flag manipulation instructions.
7) Shift and rotate instructions.
8) String Instructions.

→ 1) Data copy/Transfer Instruction:- These types of instructions are used to transfer data from source operand to destination operand. All the store, move, load, exchange, input and output instructions belongs to this category.

MOV: MOVE :- This data transfer instruction transfers data from one register / memory location to another register / memory location.

EX:- MOV AX, 5000H

PUSH :- push to stack :- This instruction pushes the contents of the specified register / memory location on to the stack.

EX:- PUSH AX
PUSH DS
PUSH [5000H]

content of location 5000H and 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 the address is formed using the current stack segment.

EX:- POP AX
POP DS
POP [5000H]

XCHG :- Exchange :- This instruction exchanges the contents of the specified source and destination operands, which may be register & one of them may be memory location.

EX:- 1) XCHG [5000H], AX → This instruction exchanges data between AX and a memory location [5000H] in the data segment.

a) XCHG BX, AX.

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 & indirectly.

EX:- IN AL, 03H → address.

out : output to the port :- This instruction is used for writing to an output port.

EX:- out 03H, AL

XLAT : Translate :- The translate instruction is used for finding out the codes in case of code conversion problems.

2) Arithmetic and logical instructions :-

These type of instructions perform Arithmetic and Logical functions.

Arithmetic instructions :-

ADD :- Addition EX:- add al, bl

ADC :- Addition with carry.

SUB - it performs subtraction operation.

SBB - Subtract with borrow.

INC - Increment

DEC - decrement

AAA - ASCII adjust for addition

AAS - ASCII adjust for Subtraction.

CMP - Compare.

DAA - Decimal adjust for addition.

DAS - Decimal adjust for Subtraction.

mul - multiplication (unsigned)

DIV - division (unsigned)

imul - multiplication (signed)

iDIV - division (signed).

Logical instructions :- It performs logical operations.

AND :- it performs and operation (And AX, BX)

OR :- It performs OR operation (OR AX, 0020)

NOT :- It performs invert operation (Not 0020)

XOR - It performs EX-OR operation. (XOR AX, BX

EX:- And AX, 0052h

AX = 0302 → 0000 0011 0000 0010

0052 → 0000 0000 0101 0010

0000 0000 0000 0010

3) Branch instructions:- (control Transfer instructions)

The control transfer instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly.

→ unconditional Branch instruction:- In case of unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition.

EX:- CALL, RET, INTN, Jump, Loop.

CALL :- unconditional call:- This instruction is used to call a subroutine from a main program.

RET : Return from the procedure

INTN:- interrupt type N

INTO :- Interrupt on overflow (OF = 1) → This command is executed, when OF = 1

JMP : unconditional Jump.

IRET : Return from ISR (interrupt service routine)

Loop : loop unconditionally.

```
EX:-   mov CX, 0005
       mov BX, 0FF7H
Label: mov AX, code1
       OR  BX, AX
       AND DX, AX
       loop label
```

The execution proceeds in sequence, after the loop is executed, CX number of times.

Conditional branch instructions:- In case of conditional branch instructions, the control will be transferred to the specified location if a particular flag satifies the conditions.

Mnemonics	operation.
→ JZ/JE	Transfer execution control to address (if ZF = 1)
→ JNZ/JNE	Transfer execution control to address (if ZF = 0)
→ JS	Transfer execution control to address (if SF = 1)
→ JNS	Transfer execution control to address (if SF = 0)
→ JO	Transfer execution control to address (if OF = 1)
→ JNO	Transfer execution control to address (if OF = 0)

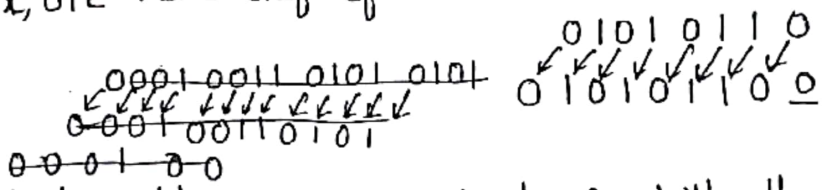
Similarly JP, JNP, JC & JNC.

4. Shift and rotate instructions:-

→ SHL/SAL: Shift Logical/Arithmetic left: These instructions shift the operand word or byte bit by bit to the left and insert zeros in the Right most significant bit.

EX:- SHL AX, 01 ← 1 time shift left

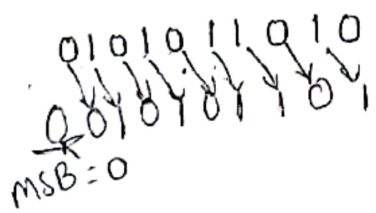
AX = 0000 0011 0101 0101



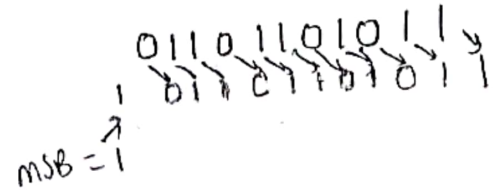
→ SHR: Shift Logical Right :- These instructions shift the operand word or byte bit by bit to the Right and insert zeros in the left most significant bit.

→ SAR:- Shift Arithmetic Right:- This instruction performs right shifts on the operand word or byte, and insert ones in the left most significant bit.

→ SHR



SAR

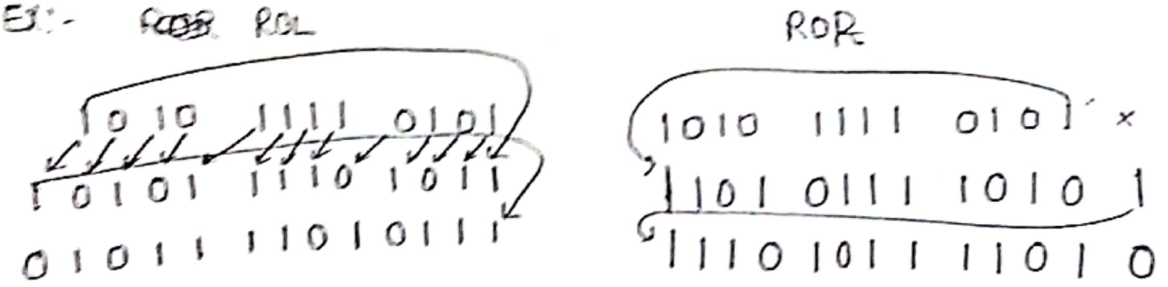


Rotate instruction :-

→ ROR :- Rotate Right without carry – This instruction rotates the contents of the destination operand to the right either by one or by the count specified.

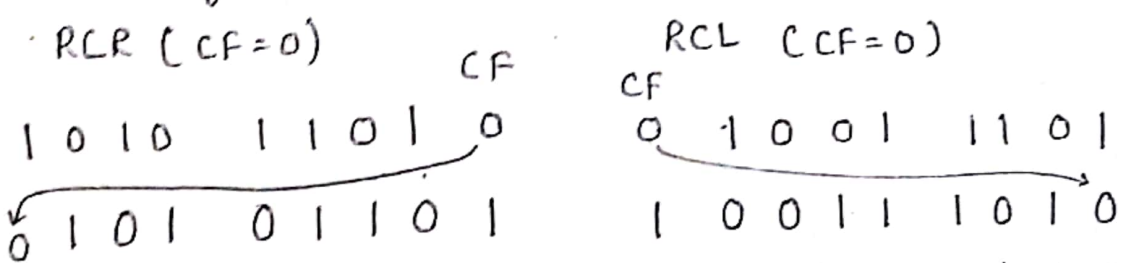
→ ROL :- Rotate left without carry :- This instruction rotates the contents of the destination operand to the left either by one or by the count specified.

Eg :-



→ RCR : Rotate Right through carry :- This instruction rotates the contents of the destination operand right by the specified count through carry flag. (CF)

→ RCL :- Rotate left through carry :- This instruction rotates the contents of the destination operand left by the specified count through carry flag (CF)



5. Flag manipulation and processor control instructions :-

The flag manipulation instruction directly modify some of the flags of 8086.

→ CLC :- Clear carry Flag
→ CMC – Complement Carry Flag
→ STC – Set carry Flag
→ CLD – Clear direction Flag.

→ STD → Set direction flag.
→ CLI – Clear Interrupt flag.
→ STI – Set interrupt flag.

6. Machine control instructions:- The machine control instructions control the bus usage and execution.

→ Wait - wait for test input pin to go low.
→ HLT - Halt the processor.
→ NOP – No operation.
→ ESC - Escape to external devices
→ LOCK - Bus lock instruction.

After executing the HLT instruction, the processor enters the halt state. when NOP instruction is executed, the processor does not perform any operation till 4 clock cycles.

ESC instruction when executed, frees the bus for an external master like a coprocessor or peripheral devices. when lock is executed, the bus access is not allowed. The wait instruction when executed, holds the operation of processor with the current status still the logic level on the $\overline{TEST}$ Pin goes low.

6. String instructions:- A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually are called as byte strings or word strings. A string two parameters are required
a) Starting & end address of the string.
b) length of the string. (it is stored in CX)

→ REP: Repeat instruction prefix:- The instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero. (REPE/REPZ)

i) Data transfer and copy instructions:-

- REPE | REPZ → repeat operation while equal | zero
REPNE | REPNZ → repeating operation while not equal | not zero
→ MOVSB | MOVSW :- move String Byte & String word :-
Suppose a string of bytes stored in a set of consecutive memory locations is to be removed to another set of destination locations.

→ CMPS: Compare String Byte & string word :- The CMPS instruction can be used to compare two strings of bytes & words. The length of the string must be stored in the register CX. If both the byte & word strings are equal, zero flag is set.

→ SCAS :- Scan String byte & string word :- This instruction scans a string of bytes & words for an operand byte & word specified in the register.

→ LODS : Load String byte & string word :- The LODS instruction loads the AL/AX register by the content of a string pointed to by DS:SI register pair.
LODSW → Load String word.

→ STOS : Store String byte & string word :- The STOS instruction stores the AL/AX register contents to a location in the string pointed ES : DI register pair.

Addressing modes of 8086 :-

Addressing mode indicates a way of locating data or operands. Depending upon the data types used in the instruction and the memory addressing modes. Types of addressing modes.

→ 1. Immediate addressing mode :- In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

EX:- mov AX, 0005H → 8 bit or 16-bit data in size.

→ 2. direct addressing mode :- In the direct addressing mode, a 16-bit memory address (offset) is directly specified in the instruction as a part of it

EX:- mov AX, [5000H] ↳ address.

→ 3. Register addressing mode :- In the register addressing mode, the data is stored in a register and it is referred using the particular register.

EX:- mov BX, AX.

→ 4. Register indirect :- Sometimes, the address of the memory location which contains data or operand is determined in an indirect way, using the offset registers. This mode of addressing is known as register indirect mode.

EX:- mov AX, [BX]

→ 5. Indexed addressing mode :- In this addressing mode, offset of the operand is stored in one of the index registers. DS is the default segment for index registers SI and DI.

EX:- mov AX, [SI]

Here, data is available at an offset address stored in SI in DS

→6. Register Relative :- In this addressing mode, the data is available at an effective address formed by adding an 8-bit & 16-bit displacement with the content of any one of the registers BX, BP, SI and DI.

EX:- MOV AX, 50H [BX]

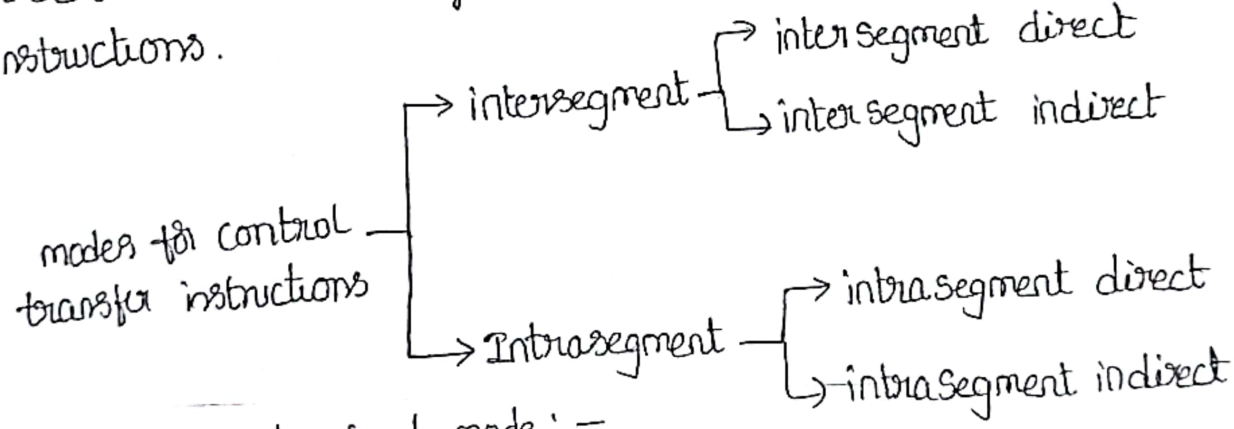
→7. Based Indexed :- The effective address of data is formed, in this addressing mode, by adding content of a base register to the content of an Index register.

EX:- MOV AX, [BX][SI].

→8. Relative Based Indexed :- The effective address is formed by adding an 8 & 16-bit displacement with the sum of contents of any one of the base registers and any one of the index registers.

EX:- MOV AX, 50H [BX] [SI].

For the control transfer instructions, the addressing modes depend upon whether the destination location is within the same segment & in a different one. Basically there are two addressing modes for the control transfer instructions.



9. Intrasegment direct mode :-

In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the

instruction as an immediate displacement value. In this addressing mode, the displacement is computed relative to the content of the instruction pointer IP.

EX:- JMP short label ;

label lies within -128 to +127 from the current IP content.

10. Intrasegment Indirect mode :- In this mode, the displacement to which the control is to be transferred, is in the same segment in which the control transfer instruction lies, but it is passed to the instruction indirectly.

EX:- JMP [BX] → Jump to effective address stored in BX.

11. Intersegment direct :- In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment.

EX:- JMP 5000H : 2000H

Jump to effective address 2000H in segment 5000H.

12. intersegment Indirect :- In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly that is contents of a memory block containing four bytes. IP (LSB), IP (MSB), CS (LSB) and CS (MSB) sequentially.

EX:- JMP [2000H]

Jump to an address in the other segment specified at effective address 2000H in DS, that points to the memory block.

Program Development and Execution:

The steps involved in Program Development and Execution of assembly language programs. Fig. 8.1 shows these steps. The left side of the figure shows the time period, at which each step in the overall process takes place.

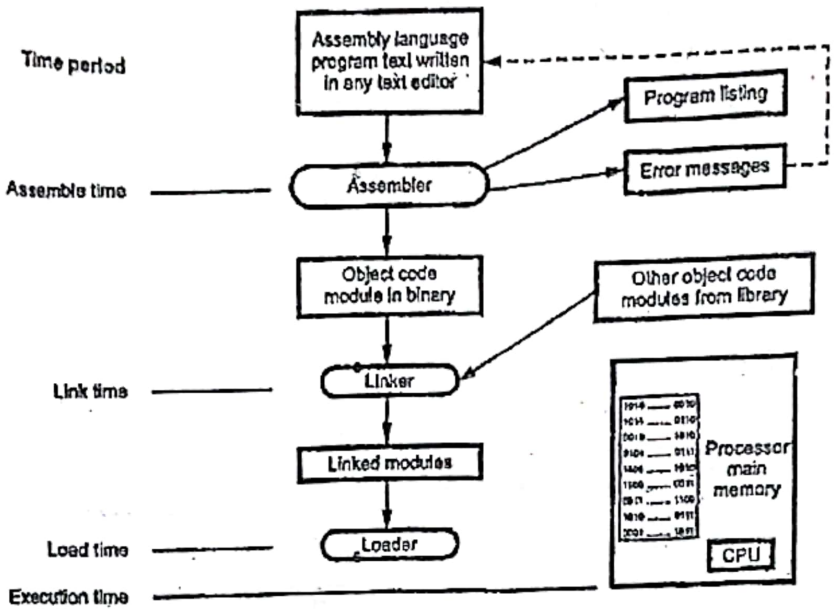


Fig. 8.1 Steps in program development and execution

The first step in the development process is to write an assembly language program. The assembly language program can be written with an ordinary text editor such as word star, edit and so on. The assembly language program text is an input to the assembler.

The assembler translates assembly language statements to their binary equivalents, usually known as object code. Time required to translate assembly code to object code is called **Assemble Time**. During assembling process assembler checks for syntax errors and displays them before giving object code module.

The object code module contains the information about where the program or module to be loaded in memory. If the object code module is to be linked with other separately assembled modules then it contains additional linkage information. At link time, separately assembled modules are combined into one single load module, by the linker.

The linker also adds any required initialization or finalization code to allow the operating system to start the program running and to return control to the operating system after the program has completed.

Most linkers allow assembly language modules to be linked with object code modules compiled from high-level languages as well. This allows the programmer to insert a time-critical assembly language routines, library modules into a program.

At load time, the program loader copies the program into the computer's main memory, and at execution time, program execution begins

Assembly Language Program Development Tools

1. Editor

- An editor is a program which allows you to create a file containing the assembly language statements for your program.

Example: PC-Write, Wordstar.

- As you type in your program, the editor stores the ASCII codes for the letters and numbers in successive RAM locations.
- When you have typed in all your program, you then save the file on the hard disk. This file is called *source file* and the extension is *.asm.*

2. Assembler

- An assembler program is used to translate the assembly language mnemonics for instructions to corresponding binary codes. When you run the assembler, it reads the source file of your program from the disk where you have saved it after editing.
- On the first pass through the source program, the assembler determines the displacement of named data items, the offset of labels, etc. and puts this information in a symbol table.
- On the second pass through the source program, the assembler produces the binary code for each instruction and inserts the offsets, etc. that it calculated during the first pass.
- The assembler generates 2 files on the floppy disk or hard disk. The first file is called object file (*.obj*).
- The second file generated by assembler is called the assembler list file and is given extension (*.lst*).

3. Linker

- A linker is a program used to join several object files into one large object file.
- The linker produces a link file which contains the binary codes for all the combined modules. The linker also produces a link map file which contains the address information about the linked files (*.exe*).

4. Locator

- A locator is a program used to assign the specific address of where the segments of object code are to be loaded into memory.
- A locator program called EXE2BIN comes with the IBM PC Disk Operating System (DOS). EXE2BIN converts a *.exe* file to a *.bin* file which has physical addresses.

5. Debugger

- A debugger is a program which allows you to load your object code program into system memory, execute the program and troubleshoot or debug it.
- The debugger allows you to look at the contents of registers and memory locations after your program runs.
- It allows you to change the contents of registers and memory locations and re-run the program.
- Some debuggers allow you to stop execution after each instruction so that you can check or alter after each register contents.
- A debugger also allows you to set a breakpoint at any point in your program. If you insert a breakpoint at any point in your program, the debugger will run the program up to the instruction where you put the breakpoint and then stop the execution.

6. Emulator:- An emulator is a mixture of hardware and software. It is used to test and debug the hardware and software of an external system. such as the proto type of a µp based instrument. part of the hardware of an emulator is a multiwire cable which connects the host system to the system being developed.

1)**Data transfer and copy instructions:-**

- **MOV** - Move Memory
- **MOVC** - Move Code Memory
- **MOVX** - Move Extended Memory
- **POP** - Pop Value From Stack
- **PUSH** - Push Value Onto Stack
- **XCH** - Exchange Bytes
- **XCHD** - Exchange Digits

2)**Arithematic instructions:-**

- **ADD, ADDC** - Add Accumulator (With Carry)
- **SUBB** - Subtract From Accumulator With Borrow
- **MUL** - Multiply Accumulator by B
- **DEC** - Decrement Register
- **DIV** - Divide Accumulator by B
- **INC** - Increment Register
- **DA** - Decimal Adjust

3)**Logical instructions:-**

- **ANL** - Bitwise AND
- **ORL** - Bitwise OR
- **RL** - Rotate Accumulator Left
- **RLC** - Rotate Accumulator Left Through Carry
- **RR** - Rotate Accumulator Right
- **RRC** - Rotate Accumulator Right Through Carry
- **XRL** - Bitwise Exclusive OR

4) **Boolean instructions**

- **CLR** - Clear Register
- **CPL** - Complement Register
- **SETB** - Set Bit
- **NOP** - No Operation

5)**Branch instructions:-**

→**Conditional branch instructions:**

- **CJNE** - Compare and Jump if Not Equal
- **DJNZ** - Decrement Register and Jump if Not Zero
- **JB** - Jump if Bit Set
- **JBC** - Jump if Bit Set and Clear Bit
- **JC** - Jump if Carry Set
- **JNB** - Jump if Bit Not Set
- **JNC** - Jump if Carry Not Set
- **JNZ** - Jump if Accumulator Not Zero
- **JZ** - Jump if Accumulator Zero

→ **Unconditional branch instructions:**

- **JMP** - Jump to Address
- **ACALL** - Absolute Call
- **AJMP** - Absolute Jump
- **LCALL** - Long Call
- **LJMP** - Long Jump
- **RET** - Return From Subroutine
- **RETI** - Return From Interrupt
- **SJMP** - Short Jump
- **SWAP** - Swap Accumulator Nibbles
- **Undefined** - Undefined Instruction

UNIT-2
ASSEMBLY LANGUAGE PROGRAMMING WITH 8086

1. Addressing modes of 8086
2. Instruction formats of 8086
3. Instruction set of 8086
4. Assembler Directives
5. Procedures
6. Macros
7. Comparison between Procedures and Macros
8. Simple ALPs – Programming examples

2.1. ADDRESSING MODES OF 8086 :

Instruction → An instruction is a command given to the microprocessor to perform a specific operation on specified data.

Addressing Mode → The method of specifying data to be operated by an instruction is called as addressing mode

The 8086 supports the following addressing modes:

1. Immediate addressing
2. Register addressing
3. Direct addressing
4. Register Indirect addressing
   - (a) Based addressing
   - (b) Indexed addressing
   - (c) Based Indexed addressing
5. Register Relative [or] Register Indirect with Displacement
   - (a) Relative Based.
   - (b) Relative Indexed
   - (c) Relative Based Indexed addressing
6. Implicit addressing
7. I/O port addressing
8. Addressing modes for Control transfer / Branch Instructions
   - (a) Intra segment mode – Direct & Indirect
   - (b) Inter segment mode – Direct & Indirect

1. Immediate addressing: The operand (or) data is available in the instruction itself.

Ex: MOV AX, 1234H
ADD AX, 4567H

2. Register addressing: The data is available in any one of the general purpose registers.

Ex: MOV AX, BX
ADD AX, BX

3. Direct addressing: The offset /effective address of the data is available in the instruction.

Ex: MOV BX, DS:[2000H]
ADD AX, DS:[3000H]

4. Register Indirect addressing:

The effective address of data is available in any one of the Base (or) Index registers

(a) Based addressing:

The effective address of data is available in Base registers - BX or BP

Ex: MOV AX, DS:[BX]

(b) Indexed addressing:

The effective address of data is available in Index registers - SI or DI

Ex: MOV AX, DS:[SI]

(c) Based Indexed addressing

The effective address of data is the sum of Base and Index registers

Ex: MOV AX, DS:[BX+SI]
MOV AX, DS:[BX][SI]

5. Register Relative addressing:

The effective address is the sum of contents of Base/Index registers and 8-bit /16-bit signed Displacement.

(a) Relative Based addressing:

The effective address is the sum of Base register and 8-bit /16-bit displacement

Ex: MOV AX, DS:[BX+25H]
MOV AX, 25H DS:[BX]

(b) Relative Indexed addressing:

The effective address is the sum of Index register and 8-bit /16-bit displacement

Ex: MOV AX, DS:[SI+25H]
MOV AX, 25H DS:[SI]

(c) Relative Based Indexed addressing:

The effective address is the sum of Base register, Index register and Displacement

Ex: MOV AX, DS:[BX+SI+25H]
MOV AX, 25H DS:[BX][SI]

6. Implicit addressing:

There are some instructions which operate on the content of Accumulator. Such instructions do not require the address of operand. This type of addressing is called as Implicit (or) Implied addressing.

Ex: DAA - Decimal Adjust Accumulator after addition
AAA - ASCII Adjust Accumulator after addition

7. I/O port addressing:

The I/O port addressing is used to access the I/O ports.
The I/O read and I/O write operations are performed through Accumulator only.

(a) Fixed port addressing: The 8-bit I/O port address is available in the instruction

```
Ex:   IN AL, 80H      ; Reads data from port address 80H to AL
      OUT 82H, AL     ; Sends data from AL to port address 82H
```

(b) Variable port addressing: The 16-bit I/O port address is available in DX register.

```
Ex:   IN AL, DX
      OUT DX, AL
```

8. Addressing modes for Control transfer / Branch Instructions

For the Control transfer instructions (or) Branch instructions such as JMP, CALL, RET,...etc, the addressing modes depend on whether the destination address lies in the same code segment (or) different code segment.

These are 2- types : (a) Intra segment mode
(b) Inter segment mode

(a) Intra-segment mode :

- In this mode, the destination address lies in the same Code segment.
- Here only IP is modified. CS remains the same.

(i) Intra-segment Direct : In this mode the instruction specifies the DISP value
destination IP = Present IP + 8-bit (or) 16-bit DISP given in instruction

```
Ex:   JMP displacement
      JMP SHORT Lable
```

Note:
(i) If the displacement is 8-bit, the destination lies within -128 bytes to +127 bytes. This type of JUMP is called as SHORT JUMP.
(ii) If the displacement is 16-bit, the destination lies within -32 K bytes to +32 K bytes This type of JUMP is called as LONG JUMP.

(ii) Intra-segment Indirect :

In this mode, the destination address is found as content of Memory location.
destination IP = 16-bit Content of memory location

Ex: JMP WORD PTR[BX]
CALL WORD PTR[BX]
destination IP ← [BX]

(b) Inter segment mode :

- In this mode, the destination address lies in different Code segment.
- It provides branching from one code segment to another code segment.
- Here both CS and IP registers will be modified.

(i) Inter segment Direct :

In this mode, the CS and IP values of destination are specified directly in the Instruction

Ex: JMP 4000:6000
CALL 4000:6000
destination CS = 4000H,
destination IP = 6000H

(ii) Inter segment Indirect:

In this mode, the CS and IP values of destination are found as content of memory locations.

Ex: JMP DWORD PTR[BX]
CALL DWORD PTR[BX]

destination IP ← [BX]
destination CS ← [BX+2]

Problem :

Calculate the offset address and 20-bit physical address for the following addressing modes. The content of different registers are given below:

DS = 4000H, ES =6000H, SS= 8000H, SP=1998H

BX = 6688H, SI= 3333H, DI= 4444H

(a) MOV AX, DS:[5060H] (or) MOV AX, [5060H] -- Direct addressing

Offset address	=	5060 H
Segment base = DS*10	=	40000 H
20-bit Physical address	=	45060 H

(b) ADD AX, DS:[BX][SI] -- Based Indexed addressing

Offset address = BX+SI	=	99BB H
Segment base = DS*10	=	40000 H
20-bit Physical address	=	499BB H

(c) XOR AX, 25H [DI] - Relative Indexed addressing

Offset address = DI+25H	=	4469 H
Segment base = ES*10	=	60000 H
20-bit Physical address	=	64469 H

(d) MOV AX, 5000 [BX] [SI] - Relative Based Indexed addressing

Offset address = BX+SI	=	99CC H
Segment base = DS*10	=	40000 H
20-bit Physical address	=	499CC H

(e) PUSH AX : It decrements SP by 2 and AX is stored at stack top pointed by SP

Here, Source is AX
Destination is Stack

SP ← SP - 2
[SP] ← AX

Destination address = SP - 2 = 1996 H
20-bit Physical address = SS*10 + 1996 =81996 H

(f) POP CX : It copies the content of stack top to CX and SP is incremented by 2

Here, Source is Stack
Destination is CX

CX ← [SP]
SP ← SP + 2

Source address = SP = 1998 H
20-bit Physical address = SS*10 + 1998 = 81998 H

2.2. INSTRUCTION FORMATS OF 8086 :

- **Instruction** is a command given to the microprocessor to perform a specific task on specified data.

- Each instruction has 2- parts

Instruction

OPCODE	OPERAND

Opcode (Operation code) → The task to be performed
Operand → The data to be operated on

- The method of specifying data to be operated by an instruction is called as addressing mode. It may immediate data/content of register /content of memory location.

- There are 6- instruction formats in 8086 instruction set. The length of an instruction may vary from 1 to 6 bytes.

(i) One byte Instruction :

- This format is only one byte long.
- It may have implied data (or) register operands.
- The least significant 3-bits of Op-code represent the register operand, if any. Otherwise, all 8-bits are Opcode bits and Operands are implied.

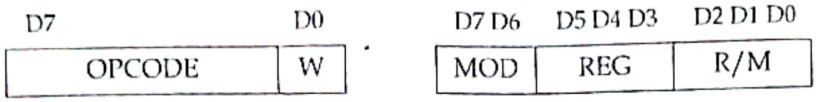
Ex: DAA ; Decial Adjsut accumulator after Addition
STC ; Set carry flag

(ii) Register to Register :

- This format is two bytes long.
- The first byte represents Op-code and width of the operand specified by ' W' bit.
- W=1 for 16-bit operand and W=0 for 8-bit operand
- The second byte represents the register operands and R/M fields

Ex: MOV AX, BX

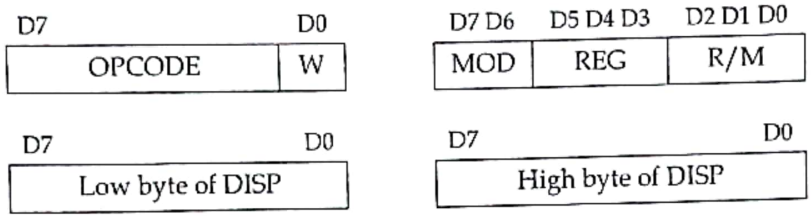
(iii) Register to/from Memory without Displacement



- This format is also two bytes long, and similar to Register to Register format, except the MOD filed.
- The first byte represents Op-code and width of the operand
- The second byte represents the register operands and R/M fields
- The MOD filed represents mode of addressing.

Ex: MOV AX, [SI]

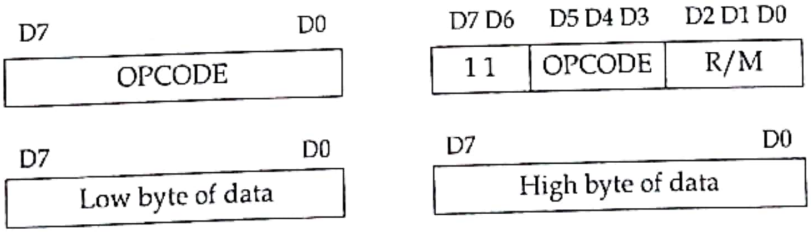
(iv) Register to/from Memory with Displacement



- This format contains one (or) two additional bytes for DISP along with 2-byte format of Register to Memory without Displacement.

Ex: MOV AX, [SI+2000H]

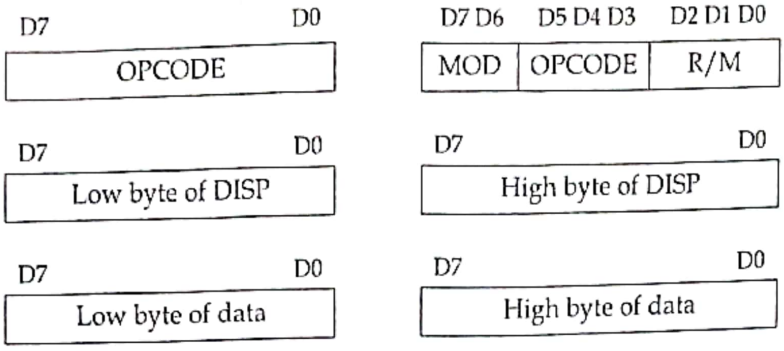
(v) Immediate operand to Register



- In this format, the first byte and 3-bits from the second byte are used to represent the Op-code. (Immediate addressing mode)
- It also contains one (or) two additional bytes of immediate data

Ex: MOV AX, 1234 H

(vi) Immediate operand to Mmeory with Displacement



- This format is 5 (or) 6 bytes long
- The first 2-bytes represent OPCODE, MOD and R/M fields.
- The next 2-bytes represent Displacement
- The last 2-bytes represent Immediate data

Registers Codes

REG	W=1	W=0
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

SREG	Segment Register
00	ES
01	CS
10	SS
11	DS

MOD, REG and R/M filed Codes

MOD / R/M	Memory operands			Reg. operands	
	00	01	10	11	
	No Disp	8-bit Disp	16-bit Disp	W=1	W=0
000	[BX+SI]	[BX]+[SI]+D8	[BX]+[SI]+D16	AX	AL
001	[BX+DI]	[BX]+[DI]+D8	[BX]+[DI]+D16	CX	CL
010	[BP+SI]	[BP]+[SI]+D8	[BP]+[SI]+D16	DX	DL
011	[BP+DI]	[BP]+[DI]+D8	[BP]+[DI]+D16	BX	BL
100	[SI]	[SI]+D8	[SI]+D16	SP	AH
101	[DI]	[DI]+D8	[DI]+D16	BP	CH
110	D16	[BP]+D8	[BP]+D16	SI	DH
111	[BX]	[BX]+D8	[BX]+D16	DI	BH

2.3. INSTRUCTION SET OF 8086 :

The 8086 Instruction set is classified as

1. Data transfer instructions
2. Arithmetic instructions
3. Logic and bit manipulation instructions
4. Branch instructions / Control transfer instructions
5. String manipulation instructions
6. Processor control instructions
   (a) Flag manipulation instructions (b) Machine control instructions

(1) Data Transfer Instructions		
1	MOV dest, source	Copies data from source to destination
2	PUSH source	Pushes the content of source onto the stack.
3	POP dest	Pop a word from stack top to destination
4	XCHG dest, source	Exchange the contents of source and destination
5	IN AL, port_addr	Read data from specified input port to Accumulator
6	OUT port_addr, AL	Send data from Accumulator to specified output port
7	LEA Reg, Operand	Load specified register with the effective address of operand
8	LDS Reg, Addr.	Load specified register and DS registers with contents of two words from the effective address
9	LES Reg, Addr.	Load specified register and ES registers with contents of two words from the effective address
10	LAHF	Load AH from lower byte of Flag
11	SAHF	Store AH to lower byte of Flag
12	XLAT	Translate byte using look-up table

(2) Arithmetic Instructions		
1	ADD dest, source	The content of source is added to the destination
2	ADC dest, source	The content of source along with carry are added to the destination
3	SUB dest, source	The content of the source is subtracted from destination.
4	SBB dest, source	The content of the source along with borrow is subtracted from destination

5	INC dest	Increases the content of destination by 1
6	DEC dest	Decreases the content of destination by 1
7	CMP dest, source	Compares destination and source operands by performing Subtraction of source from destination. Only flags are effected
8	MUL source	Unsigned multiplication of Accumulator with source
9	IMUL source	Signed multiplication of Accumulator with source
10	DIV source	Unsigned division of Accumulator by the source
11	IDIV source	Signed division of Accumulator by the source
12	CBW	Converts a signed byte in AL to a signed word in AX
13	CWD	Converts a signed word in AX to a signed word in DS:AX
14	DAA	Decimal adjust Accumulator after Addition
15	DAS	Decimal adjust Accumulator after Subtraction
16	AAA	ASCII adjust Accumulator after Addition
17	AAS	ASCII adjust Accumulator after Subtraction
18	AAM	ASCII adjust Accumulator after Multiplication
19	AAD	ASCII adjust Accumulator before Division

(3) Logical Instructions		
1	AND dest, source	Performs bitwise AND operation of Source and Destination
2	OR dest, source	Performs bitwise OR operation of Source and Destination
3	XOR dest, source	Performs bitwise Ex-OR operation of Source and Destination
4	NOT dest	Performs 1's complement of destination
5	TEST dest, source	Performs bitwise logical AND operation on the two operands. Only flags are affected. Result is not stored anywhere
6	SHL / SAR	Logical shift Left / Arithmetic shift Left (by 1 or CL)
7	SHR	Logical shift Right
8	SAR	Arithmetic shift Right
9	ROL	Rotate Left
10	RCL	Rotate Left through carry
11	ROR	Rotate Right
12	RCR	Rotate Right through carry

(4) Control transfer instructions / Branching Instructions		
1	JMP	Unconditional jump
2	CALL	Call a sub-routine
3	RET	Return to Main program
4	INT N	Software Interrupt Type N
5	IRET	Return from ISR
6	INTO	Interrupt on Overflow
7	LOOP LOOPE LOOPNE	Loop while CX ≠ 0 Loop while CX ≠ 0 and ZF = 1 Loop while CX ≠ 0 and ZF = 0
8	JCXZ	Jump if CX equals zero

(5) String manipulation instructions		
1	MOVSB/ MOVSW	Move string byte / word from DS:[SI] to ES:[DI]
2	CMPSB/ CMPSW	Compare two string bytes / words
3	LODS	Load accumulator with string byte / word from DS:[SI]
4	STOS	Store string byte / word in accumulator at ES:[DI]
5	SCAS	Compare string byte in Accumulator with ES:[DI]
6	REP REPE REPNE	Repeat while CX ≠ 0 Repeat while CX ≠ 0 and ZF = 1 Repeat while CX ≠ 0 and ZF = 0

(5) Processor control instructions (a) Flag manipulation instructions		
1	STC, CLC, CMC	Set , Clear , Complement Carry flag
2	STD, CLD	Set , Clear Directional flag
3	STI, CLI	Set , Clear Interrupt flag
4	LAHF, SAHF	Load AH from flags, store AH into flags
5	PUSHF, POPF	Push flags onto stack, pop flags off stack
(a) Machine control instruction		
1	ESC	Escape to external processor interface
2	WAIT	Wait for signal on $\overline{TEST}$ input pin active
3	LOCK	Lock bus during next instruction
4	NOP	No operation
5	HLT	Halt processor (Stops execution)

2.4. ASSEMBLER DIRECTIVES :

- Assembly language program (ALP) consists of two types of statements
  - Instructions and Directives
- Instructions are used to perform a specific operation on specified data. The instructions are translated into machine codes by the assembler.
- Directives are used to direct the assembler during assembly process, for which no machine code is generated. The Assembler directives are hints given to the assembler

S.No.	Type	Directives
1	Program Organization Directives	SEGMENT, ENDS, END ASSUME, GROUP
2	Data definition and Storage Directives	DB, DW, DD, DQ, DT
3	Alignment Directives	ORG, EVEN
4	Procedure definition Directives	PROC, ENDP
5	Macro definition Directives	MACRO, ENDM, EQU
6	Value returning attribute Directives	OFFSET, TYPE, LENGTH, SIZE, SEG
7	Data Control directives	PTR, PUBLIC, EXTRN

1. Program Organization Directives

(i) SEGMENT : This directive is used to indicate the starting of the logical segment

Syntax : Seg_name SEGMENT
Example : DATA SEGMENT

(ii) ENDS : This directive is used to indicate the ending of the logical segment

Syntax : Seg_name ENDS
Example : DATA ENDS

(iii) END : This directive is used after the last statement of the program to indicate the ending of the program

Syntax : END

(iv) ASSUME : This directive is used to inform the name of the logical segments to be used as Code segment, Data segment, Extra segment and Stack segment

Syntax : ASSUME Seg_Reg : Seg_name
Example : ASSUME CS: CODE, DS: DATA, ES: EXTRA

(v) GROUP : This directive is used to group the logical segments into one logical segment. i.e., the grouped segments will have same segment base.

Syntax : Group_name GROUP: Seg1_name, Seg2_name, ...
Example : SMALL_SYSTEM GROUP DATA, CODE, EXTRA

2. Data definition and Storage Directives

These directives are used to define the program variables and **allocate a specified amount of memory to them.** They are of type Byte, Word, Double word and Quad word.

(i) DB : This directive is used to define a variable of BYTE type.

Syntax : Var_name DB value

Examples : N1 DB 25H
N2 DB ?
ARRAY DB 10H, 20H, 30H, 40H, 50H
GRADE DB 'A'
NAME DB "MICRO"

(ii) DW : This directive is used to define a variable of WORD type (2-bytes)

Syntax : Var_name DW value

Examples : N1 DW 1234 H
ARRAY DW 1000H, 2000H, 3000H, 4000H

(iii) DD : This directive is used to define a variable of type Double word (4-bytes)

Syntax : Var_name DD value

Examples : N DD 11223344 H

(iv) DQ : This directive is used to define a variable of type Quad word (8-bytes)

Syntax : Var_name DQ value

Examples : N DD 1122334455667788 H

(v) DT : This directive is used to define a variable of type Ten bytes (10-bytes)

Syntax : Var_name DT value

Examples : N DT 11223344556677889900 H

3. Alignment Directives

These directives are used to modify the memory location counter

(i) ORG: This directive is used to set the location counter to desired value.

Syntax : ORG value

Examples : ORG 4000 H ; location counter = 4000 H
ORG $+1000 H ; location counter is incremented by 1000H

(ii) EVEN : This directive is used to increment the location counter to next Even address, if the present address is Odd. The 8086 can access a word in one bus-cycle, if the address is Even. Hence a series of words can be quickly accessed, if they are stored at Even address

Syntax : EVEN

4. Procedure definition Directives

(i) PROC : This directive is used to indicate the beginning of a Procedure. After PROC directive, the term NEAR (or) FAR is used to specify the type of the procedure.

Syntax : Procedure_name PROC NEAR/FAR
Example : Fact PROC NEAR

(ii) ENDP : This directive is used to indicate the ending of a Procedure

Syntax : Procedure_name ENDP
Example : Fact ENDP

5. Macro definition Directives

(i) MACRO : This directive is used to indicate the beginning of a Macro

Syntax : Macro_name MACRO arg1, arg2, ...
Example : Fact MACRO n

(ii) ENDM : This directive is used to indicate the ending of a Macro

Syntax : ENDM

6. Value returning attribute Directives

(i) OFFSET : This directive is used to determine the **offset address** of the variable
Example : MOV SI, OFFSET ARRAY

(ii) TYPE : This directive is used to determine the **Type** of the variable
(1- Byte , 2- Word, 4- Double word, 8- Quad word)
Example : MOV AX, TYPE ARRAY

(iii) LENGTH : This directive is used to determine the **no. of elements** in a data item
Example : MOV AX, LENGTH ARRAY

(iv) SIZE : This directive is used to determine the **no. of bytes** allocated to data item
Example : MOV AX, SIZE ARRAY

(v) SEG : This directive is used to determine the **segment base**, in which the specified data item is defined
Example : MOV AX, SEG ARRAY

7. Data control Directives

(i) PTR : This directive is used to indicate the type of memory access

Example : INC BYTE PTR [BX]
JMP WORD PTR[BX]

(ii) PUBLIC :

- This directive informs the assembler that the data items /procedures declared as PUBLIC can be accessed from any other program module.
- It helps in managing multiple program modules by sharing the global variables

```
Syntax   : PUBLIC  Var1, Var2, Var3 ....
Example  : PUBLIC  N1, N2, ARRAY
           PUBLIC  fact
```

(iii) EXTRN :

- This directive informs the assembler that the data items declared after EXTRN have already been defined in some other program module.
- Note that, only PUBLIC variables are accessible when EXTRN is used d variables can be accessed from any other program module.
- It helps in managing multiple program modules by sharing the global variables

```
Syntax   : EXTRN  Var1: Type,   Var2: Type ,  Var3: Type  ....
Example  : EXTRN  N1: Word,   ARRAY :Word
           EXTRN  fact: FAR
```

Example:

```
DATA1  SEGMENT
       N1 DB  25H
       N2 DW 1000H
       ARRAY DB 10H, 20H, 30H, 40H
DATA1  ENDS

PUBLIC  N1, N2

CODE1  SEGMENT
       ASSUME CS: CODE1,  DS: DATA1
       -----------
       -----------
       -----------
       -----------
CODE1  ENDS
       END
```

```
DATA2  SEGMENT
       N3 DB 12H
       N4 DW 2000H
       N5 DW 4000H
DATA2  ENDS

EXTRN  N1 : BYTE,  N2 : WORD

CODE2 SEGMENT
      ASSUME CS: CODE2, DS: DATA2
       ----------
       ----------

       MOV AL, N1
       MOV BX, N2

       -----------
       -----------
CODE2  ENDS
       END
```

The data items N1 and N2 are defined in Program module-1 and declared as PUBLIC. These variables can be accessed in Program module-2, by declaring them as EXTRN

Example:

Let us consider a data segment, which is defined as follows

```
DATA    SEGMENT
        ORG  4000 H
        N1  DW  1234 H
        ARRAY  DW  1000H, 2000H, 3000H, 4000H
        N2  DW  ?
DATA    ENDS
```

```
(i)  MOV  SI, OFFSET ARRAY      ; SI  = 4002H
(ii) MOV  AX, TYPE   ARRAY      ; AX = 2   (Word Type)
(iii)MOV  AX, LENGTH ARRAY      ; AX = 4   (No. of elements)
(iv)MOV   AX, SIZE   ARRAY      ; AX = 8   (No. Of bytes)
(v) MOV   AX, SEG    ARRAY      ; AX = Segment Base address of DATA
```

2.5. PROCEDURES

A procedure is a group of instructions that usually perform one task and stored in memory once, but used as often as necessary.

Advantages : (1) saves memory space
(2) makes easier to develop software
(3) we can break large program into several modules and each module can be called from main program

Disadvantages : (1) it takes some time to link from main program to procedure and Procedure to main program
(2) it uses stack memory (to store the Return address)

Two types of Procedures

(i) NEAR PROCEDURE
(ii) FAR PROCEDURE

(i) Near Procedure :

- A procedure which lies in the same code segment is called as NEAR procedure
- Since, the procedure lies in the same code segment, only IP will be modified and CS remains the same. (Intra-segment)
- The near CALL instruction is used to call a near procedure
- The Near CALL instruction performs the following **two actions**
  - It pushes the return address (only IP value) on to the stack
  - It loads IP with the offset address of procedure, so that the flow of execution is transferred to procedure

Ex:
```
CALL NEAR fact
CALL 2000H
CALL WORD PTR[BX]
```

- The RET instruction at the end of the NEAR procedure returns the flow of execution from Procedure to main program. This instruction pops the return address from stack memory.

(i) Far Procedure :

- A procedure which lies in different code segment is called as FAR procedure.
- Since, the procedure lies in different code segment, Both IP and CS will get modified. (Inter-segment)
- The FAR CALL instruction is used to call a FAR procedure

- The Far CALL instruction performs the following **two actions**
  - It pushes the return address (both CS and IP) on to the stack
  - It loads CS and IP with the address of procedure, so that the flow of execution is transferred to procedure

Ex:
```
CALL FAR fact
CALL 2000H : 5000H
CALL DWORD PTR[BX]
```

- The RET instruction at the end of the NEAR procedure returns the flow of execution from Procedure to main program. This instruction pops the return address from stack memory.

Note: The procedures are defined using directives PROC and ENDP
PROC → indicates the beginning of the procedure
ENDP → indicates the ending of the procedure

Example: **Procedure to find the factorial of given number N**

```
fact  PROC  NEAR
      MOV  AL, N        // to find the factorial of N
      MOV CL, AL
      DEC CL
      UP: MUL CL
          DEC CL
          JNZ  UP
      MOV BX, AX        // Result is copied to BX
      RET
fact  ENDP
```

PASSING PARAMETERS TO/FROM PROCEDURES :

The data values passed from main program to procedure and from procedure to main program are called as parameters. The different methods of passing parameters are

(a) Using registers
(b) Using pointes / general purpose memory
(c) Using stack memory

(a) Using registers → The parameters to be passed are stored in register and then CALL instruction is executed.

```
Ex:   MOV AL, N
      CALL fact
      MOV  RES, BX
```

In above example, the register AL is used to pass the input number N to the procedure and register BX is used to pass the result from procedure to main program.
Using this method, we can't pass more number of parameters.

(b) Using Pointers → In this method, the parameters can be directly accessed from memory using pointers from procedure.

```
Ex:   MOV  SI, OFFSET  N
      MOV  DI, OFFSET RES
      CALL fact
```

In above example, before calling the procedure, the SI and DI registers are initialized to set as source and destination pointers. Hence, the parameters can be directly accessed from memory using these pointers.
Using this method, more number of parameters can be passed by incrementing the pointers.

(c) Using Pointers → In this method, the parameters to be passed are pushed onto the stack memory, before calling the procedure.

```
Ex:   PUSH  AX
      CALL fact
      POP  BX
```

In above example, before calling the procedure, the parameters to be passed are pushed on to the stack. In procedure, we can read the parameter from stack by using POP instruction.

2.6. MACROS

- If a group of instructions are repeating again and again in the main program, the listing will be lengthy. **The process of assigning a label (or) macro name to the group of repeated instructions is called Macro.** The macro name is then used throughout the main program to refer that group of instructions.

- **During the assembly process, the assembler generates machine codes for the group of instructions and replaces the macro name with the group of instructions.**

Advantages : (1) Execution time is less (no branching takes place)
(2) It doesn't use stack memory

Disadvantages : (1) Main program length increases, because the macro name is replaced with group of instructions

- The macros are defined using directives MACRO and ENDM
  MACRO → indicates the beginning of the Macro
  ENDM → indicates the ending of the Macro

- The parameters can be directly passed to macro along with macro name

Example(1): **Macro to find the factorial of given number N**

```
FACT  MACRO  N
      MOV  AL, N
      MOV CL, AL
      DEC CL
      UP: MUL CL
          DEC CL
          JNZ  UP
      MOV BX, AX
ENDM
```

Example(2): **Macro to move a string length N of from SRC to DST**

```
MOVSTR  MACRO  SRC, DST, N
        MOV  SI, OFFSET SRC
        MOV  DI, OFFSET DST
        MOV  CX, N
        CLD
        REP :  MOVSB
ENDM
```

2.7. COMPARISION BETWEEN PROCEDURES & MACROS

S.No.	*PROCEDURES*	*MACROS*
1	Procedure is nothing but branching to a sub-routine	Macros are nothing but substitution of macro name with definition
2	Machine code will be put in memory only once, and called many times from main program	Machine code of macro are added to main program each time the macro is called
3	Program takes up less memory space	Program takes up more memory space
4	Control transfer of flow of execution is required	No control transfer of flow of execution
5	Overhead of using stack for transferring control	No overhead of using stack
6	Execution time is more	Execution time is less
7	Procedures are processed in execution time	Macros are processed in assembling time
8	Assembly time is less	Assembly time is more
9	Procedures are called by CALL instruction	Macros are called by their names
10	The directives PROC and ENP are used to define a procedure. The parameters can be passed to the procedure by using registers, pointers and stack.	The directives MACRO and ENDM are used to define a macro. The parameters can be directly passed to macro along with macro name

2.8. EXAMPLES OF ASSEMBLY LANGUAGE PROGRAMS

(1) Write an ALP to find the average of N- words which are located at ARRAY.

```
DATA    SEGMENT
        N DW 0005H
        ARRAY DW 1000H, 2000H, 3000H, 4000H, 5000H
        RES DW ?
DATA    ENDS

CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
        MOV AX, DATA
        MOV DS, AX

        MOV DX, 0000H
        MOV AX, 0000H           ; to store the sum
        MOV CX, N               ; counter
        MOV SI, OFFSET ARRAY    ; address of ARRAY

UP:     ADD AX, DS:[SI]         ; add each number to AX
        JNC down
        INC DX                  ; increment DX, only if carry

down :  INC SI
        INC SI
        LOOP UP

        MOV BX, N
        DIV BX
        MOV RES, AX
        HLT

CODE    ENDS
        END
```

(2) Write a program to move a word from location 2000:5000H to 4000:6000H.

```
CODE    SEGMENT
        ASSUME CS:CODE

        MOV AX, 2000H          ; address of source
        MOV DS, AX
        MOV SI, 5000H

        MOV AX, 4000H          ; address of destination
        MOV ES, AX
        MOV  DI, 6000H

        MOV  AX, DS:[SI]       ; Read data from source
        MOV  ES:[DI], AX       ; Store data at destination
        HLT

CODE    ENDS
        END
```

(3) An array of 16-bit numbers are located at ARRAY1. Find the 2's complement of each word and store them at ARRAY2.

```
CODE    SEGMENT
        ASSUME CS:CODE

        MOV SI,  OFFSET  ARRAY1
        MOV DI, OFFSET  ARRAY2
        MOV CX, N

  UP:   MOV  AX, [SI]          ; Read number

        NOT  AX                ; Find 1's complement
        INC  AX                ; Find 2's complement

        MOV [DI],  AX          ; Store result

        INC SI
        INC SI
        INC  DI
        INC  DI
        LOOP UP
        HLT

CODE    ENDS
        END
```

(4) An array of 8-bit numbers are located at ARRAY. Write an ALP to separate the ODD and EVEN numbers. Store the ODD numbers at ARRAY2 and EVEN numbers at at ARRAY3.

```
CODE    SEGMENT
        ASSUME CS:CODE

        MOV SI, OFFSET ARRAY    ; input array address
        MOV BX, OFFSET ARRAY2   ; to store EVEN numbers
        MOV DI, OFFSET ARRAY3   ; to store ODD numbers
        MOV CX, N               ; no. of elements

UP:     MOV AL, DS:[SI]         ; Read the number

        TEST AL, 01H
        JNZ odd_num

        MOV DS:[BX] AL          ; Store Even number
        INC DI
        JMP down

odd_num : MOV DS:[DI], AL       ; Store the Odd number
          INC BX

down: INC SI
      LOOP UP
      HLT

CODE    ENDS
        END
```

Note: To check whether the given word is Positive (or) Negative

Method(1) : By using TEST instruction : TEST AX, 8000H

- The TEST instruction performs AND operation. Here only flags are affected and result is not stored anywhere.
- The given word is ANDed with data 8000H
  if the result is zero (ZF=1) then, the given number is Positive number

Method(2) : By using CMP instruction : CMP AX, 8000H

- The given word is compared with data 8000H
  If given number < 8000H (i.e., CF=1) it is a positive number

Method(3) : By using RCL instruction : RCL AX, 1

- For negetive numbers,MSB=1
- By performing Rotate Left with carry operation, we can move the MSB to CF
- After RCL operation, if CF=1 then, it is a negative number

(5) Write a program to count the number of positive numbers and negative numbers in a given series of signed numbers

```
CODE    SEGMENT
        ASSUME CS:CODE

        MOV SI, OFFSET ARRAY
        MOV CX, N

        MOV BX, 0000H       ; to count number of positive numbers
        MOV DX,0000H        ; to count number of negative numbers

UP:     MOV AL, DS:[SI]

        TEST AL, 80H
        JNZ odd_num

        INC BX
        JMP down

odd_num : INC DX

down: INC SI

        LOOP UP
        HLT

CODE    ENDS
        END
```

(6) Write an 8086 ALP to find the largest number in given array of N-numbers

```
CODE    SEGMENT
        ASSUME CS:CODE

        MOV SI, OFFSET ARRAY
        MOV CX, N
        MOV AL,00H          ; to store the largest number

UP:     CMP AL, [SI]
        JNC down

        MOV AL, [SI]

down: INC SI
        LOOP UP

        HLT
CODE    ENDS
        END
```

(7) Write an 8086 assembly language program to generate fibonacci series

```
CODE    SEGMENT
        ASSUME CS:CODE

        MOV DI, OFFSET RES ;  to store the fibonacci series
        MOV CX, N

        MOV AL, 00H
        MOV BL, 01H

UP:     ADD AL, BL
        MOV [DI], AL
        MOV AL, BL      ; copy previous value to AL
        MOV BL, [DI]    ; copy present value to BL
        INC DI
        LOOP UP

        HLT
CODE    ENDS
        END
```

(8) Write an 8086 ALP to search a number N in given array of 10-numbers. If the number is found, store 1111H in AX. Otherwise store 0000H in AX

```
CODE    SEGMENT
        ASSUME CS:CODE

        MOV DI, OFFSET ARRAY
        MOV CX, 000AH           ; Counter = 10 numbers

        MOV AL, N               ; Number to be searched

UP:     CMP AL, [DI]
        JZ down

        INC DI
        LOOP UP
        MOV AX,0000H
        JMP exit

down:   MOV AX,1111H

exit    : HLT

CODE    ENDS
        END
```