INTERNAL ARCHITECTURE OF 8086

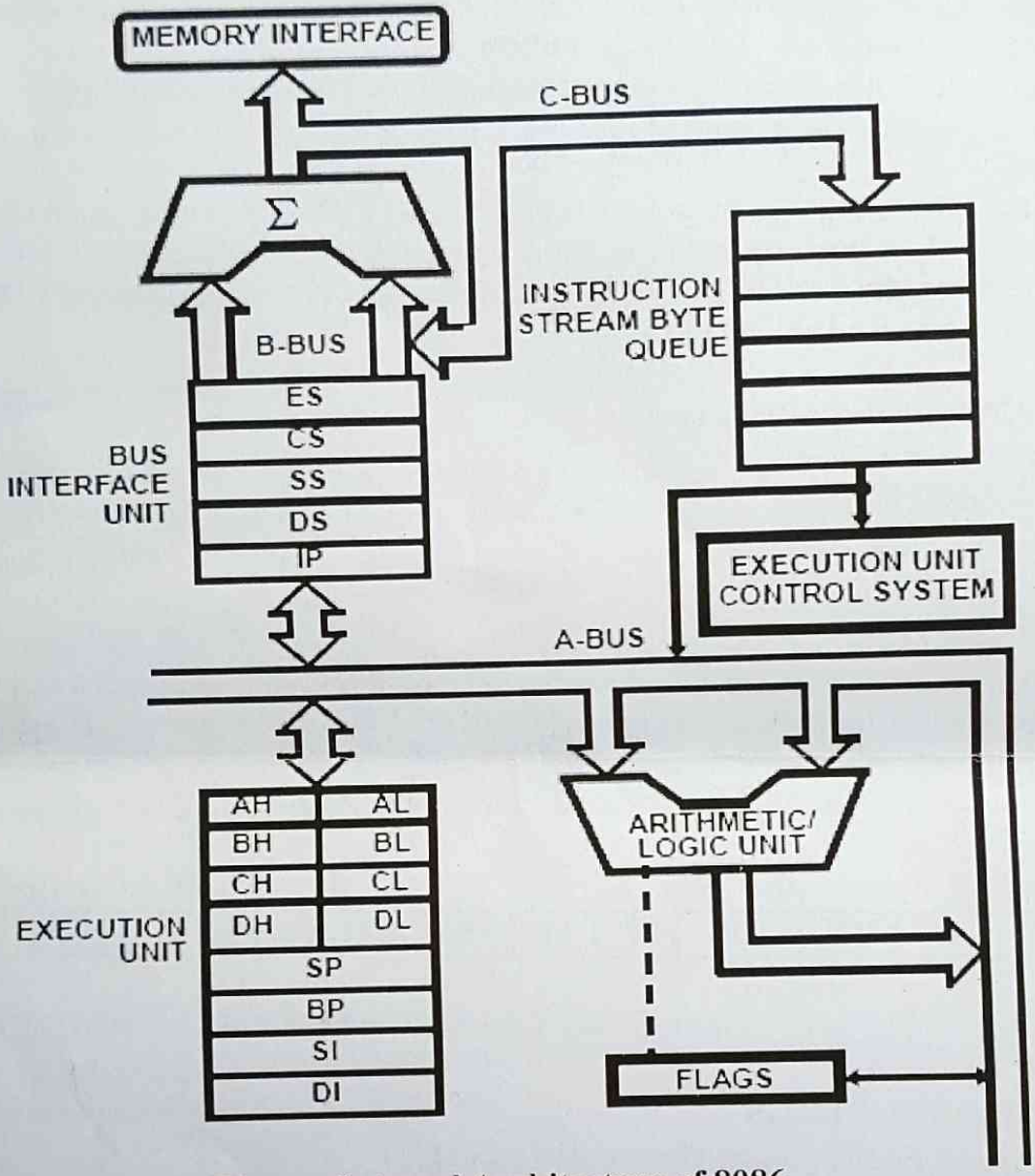

Figure: Internal Architecture of 8086

- The 8086 CPU is divided into two independent functional parts : BIU (Bus Interface Unit) and EU (Execution Unit)
- Dividing the work between these units speed up the processing.

A. THE EXECUTION UNIT (EU)

- The EU of the 8086 tells the BIU where to fetch the instructions and data from, decodes instructions and executes instructions.
- The EU has a 16 bit ALU which can add subtract, ND, OR, increment, decrement, complement or shift binary numbers.

1. GENERAL PURPOSE REGISTERS

- The EU has eight general purpose registers, labeled AH, AL, BH, BL, CH, CL, DH and DL.
- These registers can be used individually for temporary storage of 8 bit data.
- The AL register is also called accumulator
- It has some features that the other general purpose registers do not have.
- Certain pairs of these general purpose registers can be used together to store 16 bit words.
- The acceptable register pairs are AH and AL,BH and BL,CH and CL,DH and DL
- The AH-AL pair is referred to as the AX register, the BH-BL pair is referred to as the BX register, the CH-CL pair is referred to as the CX register, and the DH-DL pair is referred to as the DX register.

AX = Accumulator Register
BX = Base Register
CX = Count Register
DX = Data Register

2. FLAG REGISTER

- A Flag is a flip-flop which indicates some condition produced by the execution of an instruction or controls certain operations of the EU.
- A 16 bit flag register in the EU contains 9 active flags.
- Figure below show shows the location of the nine flags in the flag register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

Figure: 8086 Flag Register Format

U = UNDEFINED

CONDITIONAL FLAGS

CF = CARRY FLAG [Set by Carry out of MSB]
PF = PARITY FLAG [Set if Result has even parity]
AF= AUXILIARY CARRY FLAG FOR BCD
ZF = ZERO FLAG [Set if Result is 0]
SF = SIGN FLAG [MSB of Result]
OF = OVERFLOW FLAG

CONTROL FLAG

TF = SINGLE STEP TRAP FLAG
IF = INTERRUPT ENABLE FLAG
DF = STRING DIRECTION FLAG

- The six conditional flags in this group are the CF,PF,AF,ZF,SF and OF
- The three remaining flags in the Flag Register are used to control certain operations of the processor.
- The six conditional flags are set or reset by the EU on the basis of the result of some arithmetic or logic operation.
- The Control Flags are deliberately set or reset with specific instructions you put in your program.
- The three control flags are the TF, IF and DF.
- Trap Flag is used for single stepping through a program.
- The Interrupt Flag is used to allow or prohibit the interruption of a program.
- The Direction Flag is used with string instructions.

3. POINTER REGISTERS

- The 16 bit Pointer Registers are IP,SP and BP respectively
- SP and BP are located in EU whereas IP is located in BIU

3.1 STACK POINTER (SP)

- The 16 bit SP Register provides an offset value, which when associated with the SS register (SS:SP)

3.2 BASE POINTER (BP)

- The 16 bit BP facilitates referencing parameters, which are data and addresses that a program passes via the stack.
- The processor combines the addresses in SS with the offset in BP.
- BP can also be combined with DI and SI as a base register for special addressing.

4. INDEX REGISTERS

- The 16 bit Index Registers are SI and DI

4.1 SOURCE INDEX (SI) REGISTER

- The 16 bit Source Index Register is required for some string handling operations
- SI is associated with the DS Register.

4.2 DESTINATION INDEX (DI) REGISTER

- The 16 bit Destination Index Register is also required for some string operations.
- In this context, DI is associated with the ES register.

B. THE BUS INTERFACE UNIT

- The BIU send out address, fetches instructions from memory, reads data from ports and memory, and writes data to ports and memory.
- In other words BIU handles all transfers of data and addresses on the buses for the execution unit.

1. SEGMENT REGISTERS

1.1 CODE SEGMENT REGISTER (CS)

- It contains the starting address of a program's code segment.
- This segment address plus an offset value in the IP register indicates the address of an instruction to be fetched for execution
- For normal programming purpose, you need not directly reference this register.

1.2 DATA SEGMENT REGISTER (DS)

- It contains the starting address of a program's data segment
- Instruction uses this address to locate data.
- This address plus an offset value in an instruction causes a reference to a specific byte location in the data segment.

1.3 STACK SEGMENT REGISTER (SS)

- Permits the implementation of a stack in memory
- It stores the starting address of a program's stack segment the SS register.
- This segment address plus an offset value in the Stack Pointer (SP) register indicates the current word in the stack being addressed.

1.4 EXTRA SEGMENT REGISTER (ES)

- It is used by some string operations to handle memory addressing.
- ES Register is associated with the DI Register.

2. INSTRUCTION POINTER (IP)

- The 16 bit IP Register contains the offset address of the next instruction that is to execute.
- IP is associated with CS register as (CS:IP)
- For each instruction that executes, the processor changes the offset value in IP so that IP in effect directs each step of execution.

3. THE QUEUE

- While the EU is decoding an instruction or executing an instruction which does not require use of the buses, the BIU fetches up to six instructions bytes for the following instructions.
- The BIU Stores pre-fetched bytes in First in First out register set called a queue.
- When the EU is ready for its next instruction, it simply reads the instruction bytes for the instruction from the queue in the BIU.
- This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction bytes or bytes.
- Fetching the next instruction while the current instruction executes is called pipelining.

ADDRESSING MODES

- The different ways in which a processor can access data are referred to as its addressing modes.
- In assembly language statements, the addressing mode is indicated in the instruction itself.
- The various addressing modes are

1. REGISTER ADDRESSING MODE

- It is the most common form of data addressing.
- Transfers a copy of a byte/word from source register to destination register.

INSTRUCTION	SOURCE	DESTINATION
MOV AX,BX	REGISTER BX	REGISTER AX

- It is carried out with 8 bit registers AH,AL,BH,BL,CH,CL,DH & DL or with 16 bit registers AX,BX,CX,DX,SP,BP,SI and DI.
- It is important to use registers of same size.
- Never mix an 8 bit register with a 16 bit register i.e. MOV AX, BL

EXAMPLES

MOV AL, BL : Copies BL into AL
MOV ES, DS : Copies DS into ES
MOV AX, CX : Copies CX into AX

2. IMMEDIATE ADDRESSING MODE

- The term immediate implies that the data immediately follow the hexadecimal opcode in the memory.
- Note that immediate data are constant data.
- It transfers the source immediate byte/word of data in destination register or memory location.

INSTRUCTION	SOURCE	DESTINATION
MOV CH,3AH	DATA 3AH	REGISTER AX

EXAMPLES

MOV AL, 90 : Copies 90 into AL
MOV AX, 1234H : Copies 1234H into AX
MOV CL, 10000001B : Copies 100000001 binary value into CL

3. DIRECT ADDRESSING MODE

- In this scheme, the address of the data is defined in the instruction itself.
- When a memory location is to be referenced, its offset address must be specified

INSTRUCTION	SOURCE	DESTINATION
MOV AL,[1234H]	ASSUME DS=1000H 10000 H + 1234 H 11234H MEMORY LOCATION 11234 H	REGISTER AL

EXAMPLES

MOV AL, [1234H] : Copies the byte content of data segment memory location 11234H into AL.

MOV AL, NUMBER : Copies the byte content of data segment memory location NUMBER into AL.

4. REGISTER INDIRECT ADDRESSING MODE

- Register Indirect Addressing allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI and SI.
- It transfers byte/word between a register and a memory location addressed by an index or base registers.
- The symbol [ ] denote indirect addressing.

INSTRUCTION	SOURCE	DESTINATION
MOV CL,[BX]	ASSUME DS=1000H ASSUME BX=0300H 10000 H + 0300 H 10300H MEMORY LOCATION 10300H	REGISTER CL

EXAMPLES

MOV CX, [BX] : Copies the word contents of the data segment memory location addressed by BX into CX.

MOV [DI], BH : Copies BH into the data segment memory location addressed by DI.

MOV [DI], [BX] : Memory to Memory moves are not allowed except with string instructions.

5. BASE PLUS INDEX ADDRESSING MODE

- Base plus index addressing is similar to indirect addressing because it indirectly addresses memory data
- This type of addressing uses one base register (BP or BX) and one Index Register (DI or SI) to indirectly address memory.

INSTRUCTION	SOURCE	DESTINATION
MOV [BX+SI],CL	REGISTER CL	ASSUME DS=1000H ASSUME BX=0300H ASSUME SI =0200H 10000H + 0300H + 0200H 10500H MEMORY LOCATION 10500H

EXAMPLES

MOV CX, [BX+DI] : Copies the word contents of the data segment memory location addressed by BX plus DI into CX.

MOV CH, [BP+SI] : Copies the byte contents of the stack segment memory location addressed by BP plus SI into CH.

6. REGISTER RELATIVE ADDRESSING MODE

- In this case, the data in a segment of memory are addressed by adding the displacement to the content of base or an index register (BP, BX ,DI or SI).
- Transfers a byte/word between a register and the memory location addressed by an index or base register plus a displacement.

INSTRUCTION	SOURCE	DESTINATION
MOV [BX+4],CL	REGISTER CL	ASSUME DS=1000H ASSUME BX=0300H 10000H + 0300H + 4H 10304H MEMORY LOCATION 10304H

EXAMPLES

MOV [SI+4], BL : Copies BL into the data segment memory location addressed by SI plus 4

7. BASE RELATIVE PLUS INDEX ADDRESSING MODE

- The base relative plus index addressing mode is similar to the base plus index addressing mode but it adds a displacement to form a memory address.

- Transfers a byte or word between a register and the memory location addressed by a base and an index register plus a displacement.

INSTRUCTION	SOURCE	DESTINATION
MOV [BX+SI+05],CL	REGISTER CL	ASSUME DS=1000H ASSUME BX=0300H ASSUME SI =0200H 10000H + 0300H + 0200H +05H 10505H MEMORY LOCATION 10505H

EXAMPLES

MOV DH, [BX+DI+20H] : Copies the byte contents of the data segment memory location addressed by the sum of BX, DI and20H into DH

INSTRUCTION SETS

1. DATA TRANSFER INSTRUCTIONS

1.1 GENERAL PURPOSE BYTE OR WORD TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS
MOV MOV Destination,Source Eg : MOV CX,04H	Copy byte or word from specified source to specified destination.
PUSH PUSH Source Eg: PUSH BX	Copy specified word to top of stack.
POP POP Destination Eg: POP AX	Copy word from top to stack to specified location.
XCHG XCHG Destination,Source Eg: XCHG AX,BX	Exchange word or byte.

1.2 SIMPLE INPUT AND OUTPUT PORT TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS
IN IN AX,Port_Addr Eg:IN AX,34H	Copy a byte or word from specified port to accumulator.
OUT OUT Port_Addr,AX Eg: OUT 2CH,AX	Copy a byte or word from accumulator to specified port.

1.3 SPECIAL ADDRESS TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS
LEA LEA Register,Source Eg: LEA BX,PRICE	Load effective address of operand into specified register.
LDS LDS Register,Source Eg: LDS BX,[4326H]	Load DS register and other specified register from memory.

2. ARITHMETIC INSTRUCTIONS

2.1 ADDITION INSTRUCTIONS

INSTRUCTIONS	COMMENTS
ADD ADD Destination,Source Eg: ADD AL,74H	Add specified byte to byte or word to word.
ADC ADC Destination,Source Eg: ADC CL,BL	Add byte + byte + carry flag Add word+word + carry flag
INC INC Register Eg: INC CX	Increment specified byte or word by 1.
AAA	ASCII adjust after addition.
DAA	Decimal adjust after addition.

2.2 SUBTRACTION INSTRUCTIONS

INSTRUCTIONS	COMMENTS
SUB SUB Destination,Source Eg: SUB CX,BX	Subtract byte from byte or word from word.
SBB SBB Destination,Source Eg: SBB CH,AL	Subtract byte and carry flag from byte. Subtract word and carry flag from word.
DEC DEC Register Eg: DEC CX	Decrement specified byte or word by 1.
NEG NEG Register Eg: NEG AL	Form 2's complement.

CMP CMP Destination,Source Eg: CMP CX,BX CF ZF SF CX = BX 0 1 0 CX > BX 0 0 0 CX < BX 1 0 1	Compare two specified bytes or words.
AAS	ASCII adjust after subtraction.
DAS	Decimal adjust after subtraction.

2.3 MULIPLICATION INSTRUCTION

INSTRUCTIONS	COMMENTS
MUL MUL Source MUL CX	Multiply unsigned byte by byte or unsigned word by word. When a byte is multiplied by the content of AL, the result is kept into AX. When a word is multiplied by the content of AX, MS Byte in DX and LS Byte in AX.
IMUL IMUL Source IMUL CX	Multiply signed byte by byte or signed word by word.
AAM	ASCII adjust after multiplication. It converts packed BCD to unpacked BCD.

2.4 DIVISION INSTRUCTIONS

INSTRUCTIONS	COMMENTS
DIV	Divide unsigned word by byte Divide unsigned word double word by byte. When a word is divided by byte, the word must be in AX register and the divisor can be in a register or a memory location. After division AL (quotient) AH (remainder) When a double word is divided by word, the

DIV Source DIV BL DIV CX	double word must be in DX:AX pair and the divisor can be in a register or a memory location. After division AX (quotient) DX (remainder)
AAD	ASCII adjust before division BCD to binary convert before division.
CBW	Fill upper byte of word with copies of sign bit of lower byte.
CWD	Fill upper word of double word with sign bit of lower word.

3. BIT MANIPULATION INSTRUCTIONS

3.1 LOGICAL INSTRUCTIONS

INSTRUCTIONS	COMMENTS
NOT NOT Destination Eg: NOT BX	Invert each bit of a byte or word.
AND AND Destination,Source Eg: AND BH,CL	AND each bit in a byte/word with the corresponding bit in another byte or word.
OR OR Destination,Source Eg: OR AH,CL	OR each bit in a byte or word with the corresponding bit in another byte or word.
XOR XOR Destination,Source Eg: XOR CL,BH	XOR each bit in a byte or word with the corresponding bit in another byte or word.

3.2 SHIFT INSTRUCTIONS

INSTRUCTIONS	COMMENTS
SHL/SAL SHL Destination,Count CF<-MSB LSB<-0	Shift Bits of Word or Byte Left, Put Zero(s) in LSB.
SHR SHR Destination,Count 0->MSB LSB->CF	Shift Bits of Word or Byte Right, Put Zero(s) in MSB.
SAR SAR Destination,Count MSB->MSB LSB->CF	Shift Bits of Word or Byte Right, Copy Old MSB into New MSB.

3.3 ROTATE INSTRUCTIONS

INSTRUCTIONS	COMMENTS
ROL	Rotate Bits of Byte or Word Left,MSB to LS and to CF.
ROR	Rotate Bits of Byte or Word Right,LSB to MSB and to CF.
RCL	Rotate Bits of Byte or Word Left,MSB to CF and CF to LSB.
RCR	Rotate Bits of Byte or Word Right,LSB TO CF and CF TO MSB.

4. PROGRAM EXECUTION TRANSFER INSTRUCTIONS

4.1 UNCONDITIONAL TRANSFER INSTRUCTION

INSTRUCTIONS	COMMENTS
CALL	Call a Subprogram/Procedure.
RET	Return From Procedure to Calling Program.
JMP	Goto Specified Address to Get Next Instruction (Unconditional Jump to Specified Destination).

4.2 CONDITIONAL TRANSFER INSTRUCTION

INSTRUCTIONS	COMMENTS
JC	Jump if Carry Flag CF=1.
JE/JZ	Jump if Equal/Jump if Zero Flag (ZF=1).
JNC	Jump if No Carry i.e. CF=0
JNE/JNZ	Jump if Not Equal/Jump if Not Zero(ZF=0)
JNO	Jump if No Overflow.
JNP/JPO	Jump if Not Parity/Jump if Parity Odd.
JNS	Jump if Not Sign(SF=0)
JP/JPE	Jump if Parity/Jump if Parity Even (PF=1)
JS	Jump if Sign (SF=1)
LOOP	Loop Through a Sequence of Instructions Until CX=0.
JCXZ	Jump to Specified Address if CX=0.

ASSEMBLY LANGUAGE PROGRAMMING

INTRODUCTION

- Assembly Language uses two, three or 4 letter mnemonics to represent each instruction type.
- Low level Assembly Language is designed for a specific family of Processors : the symbolic instruction directly relate to Machine Language instructions one for one and are assembled into machine language
- To make programming easier, many programmers write programs in assembly language
- They then translate Assembly Language program to machine language so that it can be loaded into memory and run.

ADVANTAGES OF ASSEMBLY LANGUAGE

- A Program written in Assembly Language requires considerably less Memory and execution time than that of High Level Language.
- Assembly Language gives a programmer the ability to perform highly technical tasks.
- Resident Programs (that resides in memory while other programs execute) and Interrupt Service Routine (that handles I/P and O/P) are almost always developed in Assembly Language.
- Provides more control over handling particular H/W requirements.
- Generates smaller and compact executable modules.
- Results in faster execution.

TYPICAL FORMAT OF AN ASSEMBLY LANGUAGE INSTRUCTION

LABEL	OPCODE FIELD	OPERAND FIELD	COMMENTS
NEXT:	ADD	AL,07H	; Add correction factor

- Assembly language statements are usually written in a standard form that has 4 fields.

- A **LABEL** is a symbol used to represent an address. They are followed by colon. Labels are only inserted when they are needed so it is an optional field.

- The **OPCODE FIELD** of the instruction contains the mnemonics for the instruction to be performed. The instruction mnemonics are sometimes called as operation codes.

- The **OPERAND FIELD** of the statement contains the data, the memory address, the port address or the name of the register on which the instruction is to be performed.

- The final field in an assembly language statement is the **COMMENTS** which start with semicolon. It forms a well-documented program.

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.

2. ASSEMBLER

- An Assembler Program is used to translate the assembly language mnemonics for instruction to the corresponding binary codes.

3. LINKER

- A Linker is a Program used to join several files into one large .obj file. It produces .exe file so that the program becomes executable.

4. LOCATOR

- A Locator is a program used to assign the specific address of where the segment of object code are to be loaded into memory.
- It usually converts .exe file to .bin file.

5. DEBUGGER

- A Debugger is a program which allows you to load your .obj code program into system memory, execute program and troubleshoot.
- It allows you to look at the content of registers and memory locations after your program runs.
- It allows to set the breakpoint.

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 prototype of a Microprocessor based system.

TYPES OF ASSEMBLERS

1. **One Pass Assembler**
   - This assembler goes through or scans the assembly language program once and translates the assembly language program into binary codes.
   - This assembler has the program of defining forward references i.e a JUMP instruction using an address that appears later in the program must be defined by the programmer.

2. **Two Pass Assembler**
   - This assembler goes through or scans the assembly language program twice.
   - In the first scan, the assembler the table of symbols which consists of labels with their corresponding addresses.
   - In second scan, the assembler translates the assembly language program into the binary codes.
   - No address calculations are needed to be done by the programmer for JUMP instructions
   - It is more efficient and easier to use

MACRO ASSEMBLER

A macro assembler translates a program written in macro language into the binary codes (machine language). A macro language is the one in which all the instruction sequences can be defined using macros. A macro is an instruction sequence having specific name that appears repeatedly in a program. The macro assembler replaces a macro name with the appropriate instructions sequence, each time it encounters a macro name.

Example: a macro to add two 16 bit numbers

```
ADDITION MACRO
        MOV AX, num1
        MOV BX, num2
        ADD AX, BX
       ENDM
```

ASSEMBLER DIRECTIVES

- Assembly Language supports a number of statements that enable to control the way in which a source program assembles and lists. These Statements are called Directives.
- They act only during the assembly of a program and generate no machine executable code.
- The most common Directives are:

1. PAGE DIRECTIVE

- The PAGE Directive helps to control the format of a listing of an assembled program.
- It is optional Directive.
- At the start of program, the PAGE Directive designates the maximum number of lines to list on a page and the maximum number of characters on a line.
- Its format is

PAGE [LENGTH],[WIDTH]

- Omission of a PAGE Directive causes the assembler to set the default value to PAGE 50,80

2. TITLE DIRECTIVE

- The TITLE Directive is used to define the title of a program to print on line 2 of each page of the program listing.
- It is also optional Directive.
- Its format is

TITLE [TEXT]

TITLE "PROGRAM TO PRINT FACTORIAL NO"

3. SEGMENT DIRECTIVE

- The SEGMENT Directive defines the start of a segment.
- A Stack Segment defines stack storage, a data segment defines data items and a code segment provides executable code.
- The format (including the leading dot) for the directives that defines the stack, data and code segment are

```
.STACK [SIZE]
.DATA
    ..... Initialize Data Variables
.CODE
```

- The Default Stack size is 1024 bytes.
- To use them as above, Memory Model initialization should be carried out.

4. MEMORY MODEL DEFINTION

- The different models tell the assembler how to use segments to provide space and ensure optimum execution speed.
- The format of Memory Model Definition is

.MODEL [MODEL NAME]

- The Memory Model may be TINY, SMALL, MEDIUM, COMPACT, LARGE AND HUGE.

MODEL TYPE	DESCRIPTION
TINY	All DATA, CODE & STACK Segment must fit in one Segment of Size <=64K.
SMALL	One Code Segment of Size <=64K. One Data Segment of Size <=64 K.
MEDIUM	One Data Segment of Size <=64K. Any Number of Code Segments.
COMPACT	One Code Segment of Size < =64K. Any Number of Data Segments.
LARGE	Any Number of Code and Data Segments.
HUGE	Any Number of Code and Data Segments.

5. THE PROC DIRECTIVE

- The Code Segment contains the executable code for a program, which consists of one or more procedures, defined initially with the PROC Directive and ended with the ENDP Directive.
- Its Format is given as:

```
PROCEDURE NAME   PROC
                 ……..
                 ……..
                 ……..
PROCEDURE NAME   ENDP
```

6. END DIRECTIVE

- As already mentioned, the ENDP Directive indicates the end of a procedure.
- An END Directive ends the entire Program and appears as the last statement.
- Its Format is

END [PROCEDURE NAME]

7. THE EQU DIRECTIVE

- It is used for redefining symbolic names

EXAMPLE
DATAX DB 25
DATA EQU DATAX

8. THE .STARTUP AND .EXIT DIRECTIVE

- MASM 6.0 introduced the .STARTUP and .EXIT Directive to simplify program initialization and Termination.
- .STARTUP generates the instruction to initialize the Segment Registers.
- .EXIT generates the INT 21H function 4ch instruction for exiting the Program.

DEFINING TYPES OF DATA

The Format of Data Definition is given as
[NAME] DN [EXPRESSION]

EXAMPLES
STRING DB 'HELLO WORLD'
NUM1 DB 10
NUM2 DB 90

DEFINITION	DIRECTIVE
BYTE	DB
WORD	DW
DOUBLE WORD	DD
FAR WORD	DF
QUAD WORD	DQ
TEN BYTES	DT

- Duplication of Constants in a Statement is also possible and is given by

[NAME] DN [REPEAT-COUNT DUP (EXPRESSION)]

EXAMPLES
DATAX DB 5 DUP(12) ; 5 Bytes containing hex 0c0c0c0c0c
DATA DB 10 DUP(?) ; 10 Words Uninitialized

DATAZ DB 3 DUP(5 DUP(4)) ; 44444 44444 44444

1. CHARACTER STRINGS

- Character Strings are used for descriptive data.
- Consequently DB is the conventional format for defining character data of any length
- An Example is

  DB 'Computer City'

  DB "Hello World"

  DB "NCIT College"

2. NUMERIC CONSTANTS

#BINARY : VAL1 DB 10101010B

#DECIMAL : VAL1 DB 230

#HEXADECIMAL : VAL1 DB 23H

8086 MODES OF OPERATION

There are two available modes of operation for 8086.

1. Minimum mode of Operation
2. Maximum Mode of Operation

Minimum mode is obtained by connecting the mode selection *MN/* $\overline{MX}$ to +5V.

Maximum mode is obtained by connecting *MN/* $\overline{MX}$ to ground.

1. Minimum Mode of Operation

- In minimum mode of operation, 8086 provides all control signals needed to implement the memory and I/O interfacing.
- For minimum mode 8288 bus controller is not required
- The minimum mode signals can be divided into the following basic groups:
  a) Address/Data Bus
  b) Status Signals
  c) Control Signals
  d) Interrupt and DMA Signals

a) Address/ Data Bus

- Serves two functions
- 20 bit address bus (A_0-A_{19})
- 16 bit data lines (D_0-D_{15}) multiplexed with address lines (A_0-A_{15})

b) Status Signals

- 4 most significant address lines (A_{16}-A_{19}) are multiplexed with status signals (S_3-S_6)
- Bit S_3 and S_4 are used to specify 8086 internal segment registers.

S_4	S_3	Segment Register
0	0	Extra
0	1	Stack
1	0	Code
1	1	Data

- Bit S_5 is the logic level of the internal enable flag
- Bit S_6 is always at logic '0' level.

c) Control Signals

The following are the control signals associated with 8086 minimum mode

- **ALE (Address Latch Enable):** latch address and data
- **$\overline{\text{BHE}}$ (Bus High Enable):** $\overline{\text{BHE}}$ must be made low for read or write operation and it acts as a status signal S_7
- **$\overline{\text{M}}$/IO (Memory/IO):** IO operation if $\overline{\text{M}}$/IO=1
  Memory operation if $\overline{\text{M}}$/IO=0
- **DT/ $\overline{\text{R}}$ (Data Transmit/Receive):** Data Transmitting if DT/ $\overline{\text{R}}$ =1
  Data Receiving if DT/ $\overline{\text{R}}$ =0
- **$\overline{\text{RD}}$ (Read):** indicates read bus cycle when $\overline{\text{RD}} = 0$
- **$\overline{\text{WR}}$ (Write):** indicates write bus cycle when $\overline{\text{WR}} = 0$
- **$\overline{\text{DEN}}$ (Data Enable):** tells the external device when to put data
- **READY :** used to insert wait states into the bus

d) Interrupt and DMA Signals

- The key interrupt signals are **INTR** (Interrupt Request) and **INTA** (Interrupt Acknowledge)
- The DMA interface of the 8086 minimum mode consists of **HOLD** and **HLDA** signals

Minimum Mode Timing Diagrams

a) Read Cycle Timing Diagram for Minimum Mode (Input)

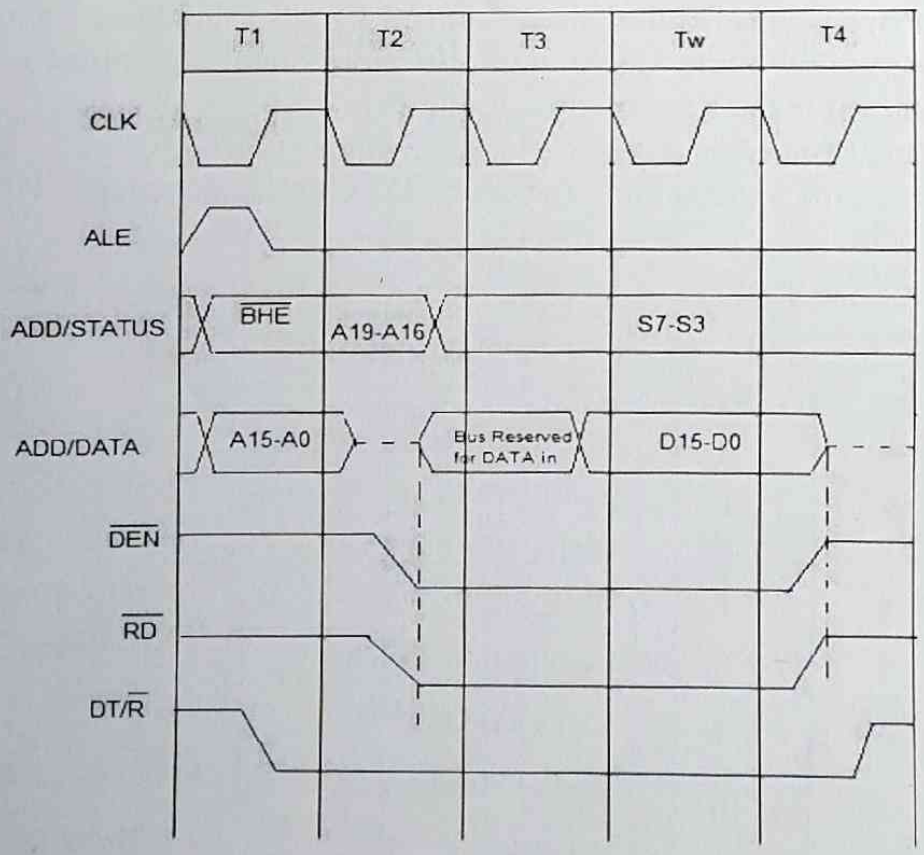

Fig: Read Cycle Timing Diagram

b) Write Cycle Timing Diagram for Minimum Mode (Output)

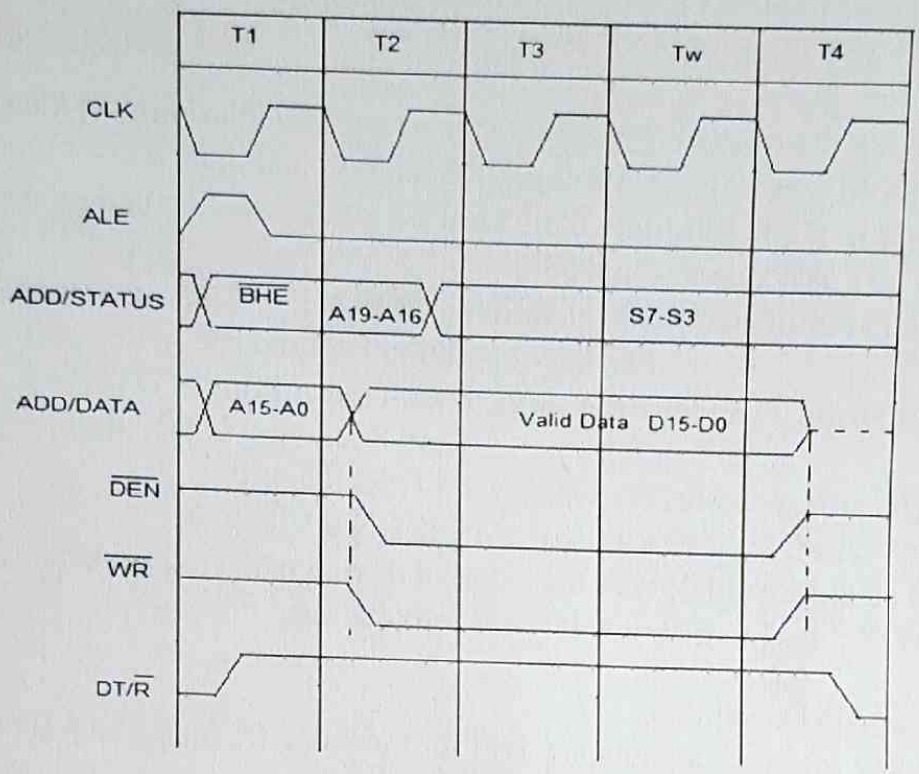

Fig: Write Cycle Timing Diagram

2. Maximum Mode

- In maximum mode, the 8086 is operated by connecting the MN/$\overline{MX}$ pin to ground.
- In this mode, the processor derives the status signals S_2, S_1, S_0. Another chip called Bus Controller derives the control signals using these status signals.
- In maximum mode, there may be more than one microprocessors (co-processors) in the system configuration.
- The Bus Controller receives the three status signals S_2, S_1, S_0 from 8086 and generates the signals that are needed to control the memory, IO and Interrupt Interfaces.

Status Inputs			CPU Cycles	8288 Command
$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$		
0	0	0	Interrupt Acknowledge	$\overline{INTA}$
0	0	1	Read I/O Port	$\overline{IORC}$
0	1	0	Write I/O Port	$\overline{IOWC}$, $\overline{AIOWC}$
0	1	1	Halt	None
1	0	0	Instruction Fetch	$\overline{MRDC}$
1	0	1	Read Memory	$\overline{MRDC}$
1	1	0	Write Memory	$\overline{MWTC}$, $\overline{AMWC}$
1	1	1	Passive	None

- **$\overline{\text{INTA}}$** is used to issue interrupt acknowledge pulses to the interrupt controller or to the interrupting device
- **$\overline{\text{IORC}}$** (IO Read Command), **$\overline{\text{IOWC}}$** (IO write Command) enable an IO interface to read or write data from or to the address port
- **$\overline{\text{MRDC}}$** (Memory Read Command), **$\overline{\text{MWTC}}$** (Memory Write Command) are used to read from or write into memory locations
- For both IO and Memory Write Command signals, the advance signals namely **$\overline{\text{AIOWC}}$** (Advance IO Write Command) and **$\overline{\text{AMWTC}}$** (Advance Memory Write Command) are available.

Maximum Mode Timing Diagrams

a) Read Cycle Timing Diagram

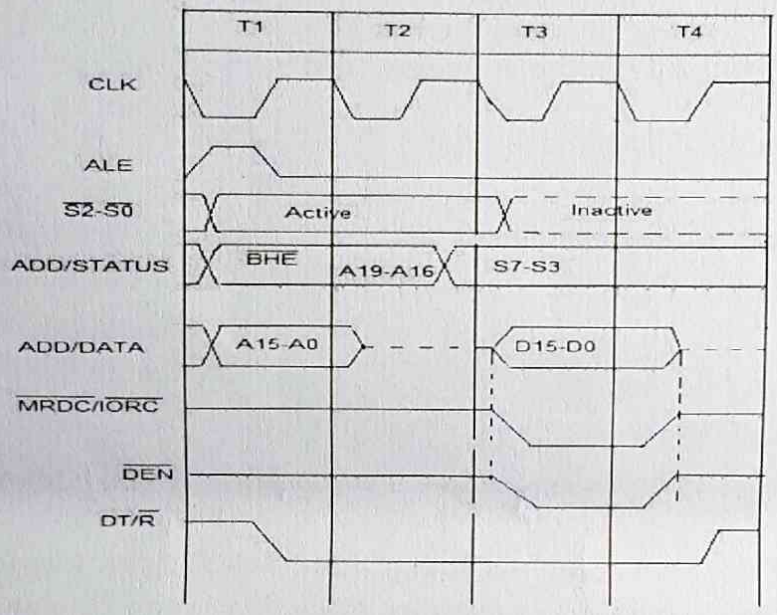

Fig: Read Cycle Timing Diagram

b) Write Cycle Timing Diagram

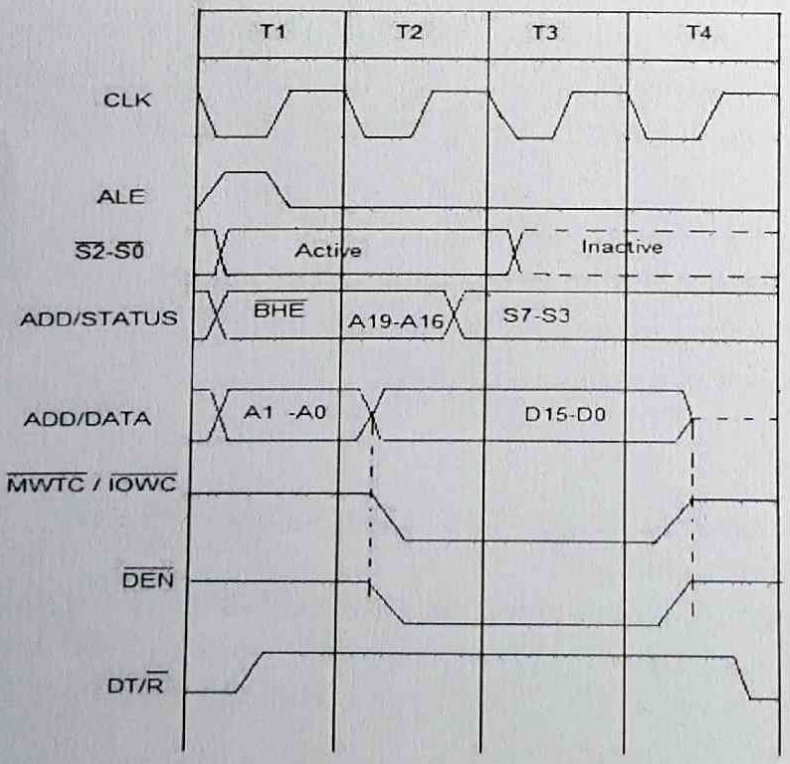

Fig: Write Cycle Timing Diagram

MODULAR PROGRAMMING

1. LINKING AND RELOCATION

Program is composed from several smaller modules. Modules could be developed by separate teams concurrently. The modules are assembled producing .OBJ modules (Object modules). The DOS linking program links the different object modules of a source program and function library routines to generate an integrated executable code of the source program.

The main input to the linker is the .OBJ file that contains the object modules of the source programs. The linker program is invoked using the following options.

C> LINK

or

C>LINK MS.OBJ

The .OBJ extension is a must for a file to be accepted by the LINK as a valid object file.

The first object may generate a display asking for the object file, list file and libraries as inputs and an expected name of the .EXE file to be generated. The output of the link program is an executable file with the entered filename and .EXE extension. This executable filename can further be entered at the DOS prompt to execute the file. Linking is necessary because of the number of codes to be linked for the final binary file.

The linked file in binary for ***run*** on a computer is commonly known as executable file or simply '.exe.' file. After linking, there has to be re-allocation of the sequences of placing the codes before actually placement of the codes in the memory. The loader program performs the task of reallocating the codes after finding the physical RAM addresses available at a given instant. The ***loader*** is a part of the operating system and places codes into the memory after reading the '.exe' file. This step is necessary because the available memory addresses may not start from 0x0000, and binary codes have to be loaded at the different addresses during the run. The loader finds the appropriate start address.

In a computer, the loader is used and it loads a program that is ready to run, into a section of RAM. A program called *locator* reallocates the linked file and creates a file for permanent location of codes in a standard format.

2. STACK

- The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU.
- It is a top-down data structure whose elements are accessed using the stack pointer (SP) which gets decremented by two as we store a data word into the stack and gets incremented by two as we retrieve a data word from the stack back to the CPU register.

- The process of storing the data in the stack is called 'pushing into' the stack and the reverse process of transferring the data back from the stack to the CPU register is known as 'popping off' the stack.
- The stack is essentially *Last-In-First-Out* (LIFO) data segment. This means that the data which is pushed into the stack last will be on top of stack and will be popped off the stack first.
- The stack pointer is a 16-bit register that contains the offset address of the memory location in the stack segment. Stack Segment register (SS) contains the base address of the stack segment in the memory.

3. PROCEDURES/SUBROUTINES

- Procedure or a subroutine or a function is a key concept for modular programming, the essential way to reduce complexity.
- A procedure is a reusable set of instructions that has a name.
- Only one copy of the procedure is stored in the memory; and it can be called as many times as needed.
- As only one copy is stored, it saves memory; but has execution time overhead for the CALL and RETURN operations.
- CALL transfers control to the procedure just like in JUMP; but unlike a JUMP, procedure has a RETURN instruction which returns control to the instruction following the CALL instruction
- In order to implement such a return, the necessary information is stored on a stack, before transferring control to the procedure.
- In program, procedure starts with **PROC** directive and ends with **ENDP** directive.
- **PROC** directive is followed by the type of procedure: **NEAR** or **FAR**.

1.3.1 Calling a Procedure

- CALL instruction followed by procedure name is used in a program to call a procedure.
- Procedure calls are of two types: Near CALL and Far CALL.

Near CALL

- A procedure may be in the same code segment as that of the main program (**Intra segment**). In such a case, we specify only IP. This is known as **NEAR CALL.**

Example:

```
Main program:
....
.....
CALL DISPLAY     ; calling a procedure. Transfer control to procedure named as DISPLAY
.....
..... Procedure Definition.....
DISPLAY PROC NEAR
MOV DX, OFFSET string1
MOV AH, 02H
INT 21H
RET
DISPLAY ENDP
```

FAR CALL

- A procedure may be in a different code segment (**Inter segment**). In such a case, we need to specify both IP and CS (directly or indirectly). This is known as a **FAR CALL.**

Example:

```
DISPLAY PROC FAR
. . . . . .
DISPLAY ENDP
```

Now, a CALL to DISPLAY is assembled as FAR CALL.

1.3.2 Return from Procedure

- We use a **RET** instruction to return from the called procedure.
- The control returns to the instruction following the CALL instruction in the calling program. Corresponding to the two varieties of CALL instructions (near & far), two forms of RET instructions (near & far) exist.
- Near RET instruction POPS 16-Bit word from the stack and places it in the IP.
- Far RET instruction POPS two 16-Bit words from the stack and places them in IP & CS.
- In ALP, RET is written within the procedure, before the ENDP directive and the Assembler will automatically select the proper (Near or Far) RET instruction.

4. MACRO

- **MACRO** is a group of instructions with a name.
- When a macro is invoked, the associated set of instructions replaces the macro name in the program. This macro expansion is done by a Macro Assembler and it happens before assembly. Thus the actual Assembler sees the expanded source.

MACRO Definition:

A macro has a name. The body of the macro is defined between a pair of directives, **MACRO** and **ENDM**.

Examples of Macro Definitions:

```
DISPLAY MACRO          ; Definition of a Macro named DISPLAY
MOV DX, OFFSET string1
MOV AH, 09H
INT 21H
ENDM                                ; end of Macro
```

MACROS Vs PROCEDURES

1) Procedure:

- Only one copy exists in memory. Thus memory consumed is less.
- Called when required
- Return address (IP or CS:IP) is saved (PUSH) on stack before transferring control to the subroutine through CALL instruction. It should be popped (POP) again when control comes back to calling program with RET instruction.
- Execution time overhead is present because of the call and return instructions.
- If more lines of code, better to write a procedure than a macro.

2) Macro:

- When a macro is invoked, the corresponding code is inserted into the source. Thus multiple copies of the same code exist in the memory leading to greater space requirements.
- However, there is no execution overhead because there are no additional call and return instructions.
- No use of stack for operation.
- Good if few lines of code are in the Macro body.