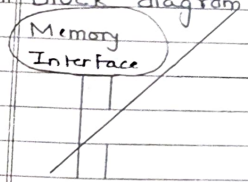
CHAPTER 3: 8086 MP

Basic Features

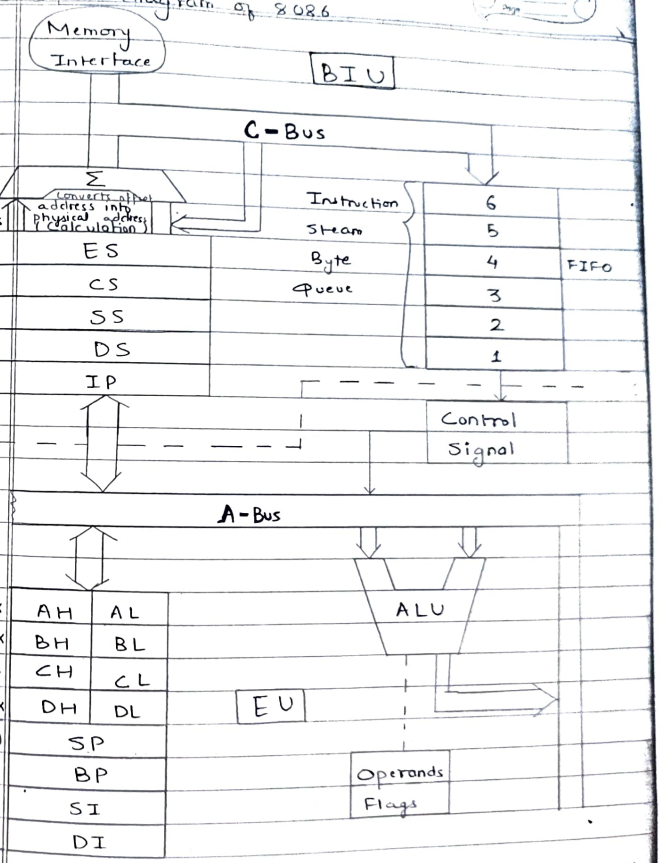
- A 16-bit MP introduced in 1978.
- 40-pin DIP
- +5V power supply
- Data bus width → 16-bit & address bus width → 20bit
  Total Addressable Memory = $2^{20} \approx 1$MB.
- 8086 block is divided into two parts: BIU & EU.
- Memory Segmentation
- Pipelining of Instruction
- Operating modes: 1. Minimum mode (Single Processor)
  2. Maximum mode (Multi Processor)
- 256 vectored interrupts

Block diagram of 8086



- Physical Address → 20-bit
- Effective / Offset Address → 16-bit

Block diagram of 8086



BIU → Bus Interface Unit

EU → Execution Unit

Features of BIU

1. It generates the physical address (20-bit)
2. It sends out addresses to the memory.
3. It fetches instruction from memory.
4. It sends out data to the memory.
5. It writes data in the output port.
6. Fetched instruction are stored in 6-byte queue that works in FIFO basis.
7. It is responsible to decode the instructions & generates the necessary control signals.

Features of EU

1. It is responsible for arithmetic and logic operati
2. It receives instruction from BIU.
3. General Purpose registers are used for the storage of data.
4. 9-Flags are associated with ALU.
5. Pointer and Index register used to hold the offset address.

Register Organisation of 8086:

8086 has got powerful set of registers as;

1.	General Purpose Registers	16-bit	AX, BX, CX, DX
		8-bit	AH, AL, BH, BL, CH, CL, DH, DL
2.	Pointer Registers	16-bit	SP (Stack Pointer) BP (Base Pointer)
3.	Index Registers	16-bit	SI (Source Index) DI (Destination Index)
4.	Segment Register	16-bit	CS (Code Segment) DS (Data ") ES (Extra ") SS (Stack ")
5.	Instruction Register	16-bit	IP (Instruction Pointer)
6.	Flag Register	16-bit	9 Flags

- General Purpose Registers are 16-bit or 8-bit registers as shown used for the storage of data
- Pointer and index registers are 16-bit registers used to hold the 16-bit offset address. These registers are associated with memory segment.
- Segment registers are used to hold the upper 16-bit of the starting address of the memory segment.
- Flag register are used to indicate the operation ongoing inside ALU.

Memory Segmentation

- 8086 has got 1MB memory that ranges from 00000H and to FFFFFH. These portion of memory is divided into four segment each having 64KB.
- The ease of memory segmentation is that we can address the memory of 8086 16-bit address although the phyical address is 20-bit
- 16-bit segment registers (DS, CS, SS & ES) are associated with thwe memory segment that holds the upper 16 bit of the starting address of the memory segment

↳ Data Segment → Data are operated in this memory
Code " → Instructions / opcode " " " "
Stack " → Stack operation
Extra " → Data / Strings Operations.

↳ Pointer & Index registers within the memory segment gives the offset address.
i.e CS → IP
DS → SI / DI
SS → SP or BP
ES → DI

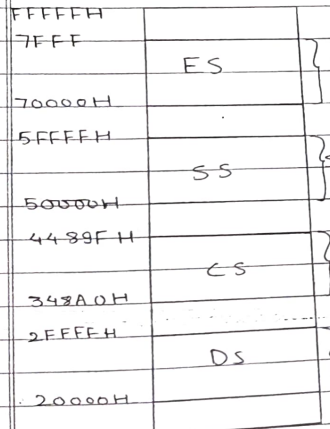


Fig: Memory Segmentation.

4489FH
Physical Address Calculation 348A0H
IP
CS
CS = 348AH
IP = 1234H
PA = (Segment Address) X 10 + Offset Address
= (348A) X 10 + 1234H
= 348A0 + 1234
= 35AD4H

Flag Register of 8086:

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF

↳ Flags are associated with ALU that indicates the status / operation.

Flags of 8086

Conditional Flag
- Sign Flag (SF)
- Zero Flag (ZF)
- Auxillary Carry Flag (AF)
- Parity Flag (PF)
- Carry Flag (CF)
- Overflow Flag (OF)

Control Flag
- Interrupt Flag (IF)
- Trap Flag (TF)
- Direction Flag (DF)

Control Flags
1. Interrupt Flags:
It is interrupt enable / disable flag. If it is set to 1, the maskable INTR of 8086 is enabled and if it is 0, the INTR is disabled. It can be set / Reset by using instruction
STI → Set Interrupt
CLI → Clear Interrupt

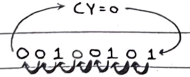
2. Direction Flag:
It is used for string operations. If it is set to '1', string bytes are accessed from high memory address to low memory address. When it is '0', the string bytes are accessed from low memory to high memory address.

3. Trap Flag:
It allows users to execute one instruction of a program at a time of debugging. Also known as single step control flag. When it is set to '1', program can be run in single step mode.

WAP to find the number of 1's present in a byte of data.

→ Suppose the data is 25 = 00100101

```
        MVI  A, 25H
        MVI  C, 08H
        MVI  B, 00H
loop:   RAL
        JNC here      ; JC for no. of 0's
        INR B
here:   DCR C
        JNZ loop
        MOV A, B
        STA 2050H
        HLT
```



Instruction Sets of 8086

8086 μP has got following sets of instructions:
1. Data Transfer Instruction
2. Arithmetic Instruction
3. Logic / Bit Manipulation Instruction
4. Program Execution Transfer Instruction
5. String Instruction
6. Process Control Instruction

1. Data Transfer Instruction
↳ Used to transfer data from source to destination
↳ Operand can be constant, memory location register or I/O port address.
It consists of the following:

@ MOV Des, Src:
eg: MOV CX, 0025H
MOV AL, BL
MOV BX, [0205H]

ⓑ PUSH Operand:
It pushes the operands onto the top of stack.
Eg: PUSH BX

ⓒ POP Des:
It pops the operand from the top of the ~~state~~ stack to Des.
↳ Des can be general purpose registers, segment registers (except CS). or memory location.
Eg: POP AX

(d) XCHG Des, Src :
→ This instruction exchanges src with Des
→ It can't exchange, has memory locations directly
Eg: XCHG, DX, AX

(e) IN Accumulator, Port Address :
→ It transfers the operand from specified port to the Accumulator.
Eg: IN AX, 0028H.

(f) OUT port address, Accumulator :
→ It transfers operand from Accumulator to the specified port.
Eg: OUT 0028H, AX.

(g) LEA Register, Src :
→ It loads the 16-bit register with the offset address of the data specified by the src.
Eg: LEA BX, [DI].

(h) LDS Des, Src :
→ It loads the 32-bit pointer from memory source to destination register & DS (Data Segment)
→ The offset is placed in the destination register and the segment is placed in DS.
Eg: LDS BS, [0301H].

(i) LES Des, Src :
→ It loads the 32-bit pointer from memory source to destination register and ES.
→ The offset is placed to des register and the segment is placed in ES. Eg: LES BX, [0301H]

(j) LAHF
→ It copies the lower byte of flag register to AH.

(k) SAHF
→ It copies the content of AH to lower byte of flag.

(l) PUSHF
→ It pushes flag register onto the top of stack.

(m) POPF
→ It pops the top of stack into the flag register.

2. Arithmetic Instructions

(a) ADD Des, Src
→ It adds a byte to byte or word to word.
→ It affects AF, CF, OF, PF, SF, ZF flags.
Eg:

ADD AL, 72H	AL ← AL+72H
ADD DX, AX	
ADD AX, [BX]	AX ← AX + (BX)

↳ Memory bhitrako data

(b) ADC Des, Src
→ It adds two operands with CF.
→ It affects AF, CF, OF, PF, SF, ZF flags.
Eg: ADC AL, 72H
ADC DX, AX
ADD AX, [BX]

(c) SUB Des, Src

→ It subtracts a byte from byte or word from word

→ It affects AF, CF, OF, PF, SF, ZF flags.

→ CF acts as a borrow flag.

Eg:

```
SUB AL, 72H
SUB DX, AX
SUB AX, [BX]
```

(d) SBB Des, Srz

→ It subtracts the two operand and also borrow from the result.

→ It affects AF, CF, OF, PF, SF, ZF Flag.

Eg:

```
SBB AL, 72H
SBB DX, AX
SBB AX, [BX]
```

(e) INC Src

→ It increments the byte or word by 1.

→ The operand can be register or memory.

→ It affects AF, OF, PF, SF, ZF flags.

→ CF is not affected.

Eg: INC b1

(f) Dec Src

→ It decrements the byte / word by 1.

Eg: Dec bX

(g) AAA (ASCII Adjust After Addition).

→ The data enkred from the terminal is in ASCII format

→ In ASCII 0-9 are represented by 30H - 39H.

→ This instruction allows us to add the ASCII codes

(h) AAS

(i) AAM

(j) AAD

(k) NEG Src

→ It creates 2's complement of the given number / src.

(l) MVL Src

→ It is an unsigned multiplication instruction.

→ It multiplies two bytes to produce a word or two word to produce a double word.

AX = AL * Src.

Ax = AX * Src

→ This instruction assumes one of the operands in AL or AX.

→ Src can be register or memory.

(m) DIV Src

→ It is unsigned division.

→ The operand is stored in AX, Divisor is Src and the result is stored as:

AH = remainder,

AL = quotient

3 Logic / BIT Manipulation Instruction

(a) NOT Src
→ It complements each bit of src to produce 1's complement of specified operand.
→ The operand can be register or memory.

(b) AND Des, Src
→ AND operator of Src and Des
→ Src can be immediate number, register or memory.
→ Des can be register or memory.
→ Both operands can't be memory simultaneously.
Eg: AND AL, BL.

(c) OR Des, Src

(d) XOR Des, Src

(e) SHL Des, Count
→ It shifts bits of byte/word left by count.
→ It puts zeros in LSB.
→ MSB is shifted into Carry Flag.

(f) SHR Des, Count
→ It shifts bits of byte/word right by count
→ It puts zeros in MSB.
→ LSB is shifted into CF.

(g) ROL Des, Count
→ It rotates bits of byte/word left by count.
→ MSB is transferred to LSB and CF.

(h) ROR Des, Count
→ Rotate right by count
→ LSB is transferred to MSB and CF

(i) CMP Des, Src
→ It compares two specified bytes/words
→ Src, Des can be constant, register or memory.
→ Flags are modified according to the result
⇒ See 8085 (For Flags condn)

4. Program Execution Transfer Instruction

(a) CALL Des:
→ This instruction is used to call a subroutine or procedure.
→ The address of next instruction after CALL is saved onto stack.

(b) RET
→ It returns the control from procedure to CALLing program.

(c) JMP Des
→ Unconditional jump from one place to another.

(d) JXX Des
→ Conditional Jump Instructions (that effects the Flags)
→ The following table shows the list of instruction:

Conditional Jump Instructions	Condition
JA → Jump if above	CF → 0, ZF → 0
JAE → Jump if above or equal	CF → 0
JB → Jump if below	CF → 1
JBE → Jump if below or equal	CF → 1, ZF → 1
JC → Jump if carry	CF → 1
JNC → Jump if no carry	CF → 0
JZ → Jump if zero	ZF → 1
JNZ → Jump if no zero	ZF → 0
JPE → Jump if parity even	PF → 1
JPO → Jump if parity odd.	PF → 0

B. LOOP Des
→ This is looping instruction and the number of loop/iteration is placed in cx register
→ With each iteration, the content of cx are decremented.

5. String Instructions
→ String in assembly language is just a sequentially stored bytes/words.
→ By using the instructions, the size of the program is considerably reduced.
→ It consists of the following instructions:

@ CMP Des, Src:
It ~~consis~~ compares the string bytes or words.

(b) SCAS String:
It scans a string. It compares the string with byte in AL or with word in AX.

(c) MOVS/MOVB/MOVSW:
→ It causes moving of bytes or word from one string to another. In this instruction the source is in data segment and the destination is in extra segment.
→ SI and DI store the offset values for the source & destination string.

(d) Rep (Repeat):
→ This is an instruction prefix.
→ It causes the repetition of instruction until cx becomes zero.

(e) Processor Control Instruction:
These instructions controls the processor itself. Some of these instructions are:
I. STC : It sets the CF to 1.
II. CLC : It clears the CF to 0.
III. CMC : It complements the CF.
IV. STD : It sets the direction Flag (DF) to 1.
V. CLD : It clears the DF to 0.

5th imp

Assembly Language Programming of 8086

* Sample Program (To display a string "Hello World")

Assembler Directive

```
.Model small
.stack 100h          → stack segment ko size 100
.Data
    String db "Hello World!$"
.Code
main proc
.STARTUP { mov ax, @DATA ; @DATA is predefined symbol for initial address of DS
           mov ds, ax    ; initialize data segment.
           mov dx, offset string }; load the offset address into dx
           OR
           LEA dx, string
           mov ah, 09h ; ax = 09h for string display until $.
           Int 21h ; DOS interrupt function.
.EXIT    { mov ax, 4c00h ; End request with AX = 4c00h
           Int 21h       ; return control to OS
main endp      ; end procedure
end main       ; end program
```

ALP Development Tools:

1. Editor → .asm
2. Assembler → One pass Assembler
   → Two pass Assembler (.obj)
3. Linker (.obj)
4. Debugger → .exe
5. Emulator (combn of ~~btw one~~ hardware and software)

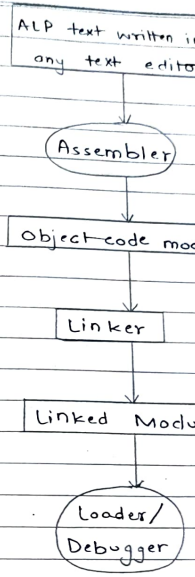


Fig: Flowchart for the development of the ALP tools.

Qn. Differentiate one pass assembler and two pass assembler.

Imp # Assembler Directives

An ALP contains two types of statements: instruction and assembler directives. The instructions are translated into machine codes whereas directives are not converted into machine codes. The directives are statements to give directions to the assembler to

perform the task of assembly process. They indicate how an operand or a section of a program is to be processed by the assembler. The assembler supports directives for data definitions, segment organizations, procedure, macro definitions etc.

↳ Some imp directives are;

① .DATA : It provides shortcut in definition of the data segment.

② DB (Define Byte) : The directive DB defines a byte type variable. It directs the assembler to reserve one byte of memory and initialize the byte with the specified value.

→ It can define single or multiple variables.

Example :

```
TEMP    DB  5
TEMP    DB  ?
TEMP    DB  11,22,33,44.
```

③ DW (Define Word) → 2 bytes

④ DD (Define double word) → 4 bytes

⑤ DQ (Define Quad word) → 8 bytes

⑥ .CODE : This provide shortcut in definition of the code segment. It is basically specified to distinguish different code segment when there are multiple CS in the program.

Eg: .CODE [name]
optional

⑦ PROC : It tells the start of the procedure or subroutine.

⑧ SEGMENT :

It indicates the beginning of a logical segment. The name of the segment is written before the directive SEGMENT.

Eg: Program SEGMENT
Segment name

⑨ ENDS :

It informs the assembler; the end of the segment. It is used with the segment directive.

⑩ ENDP :

It informs the assembler the end of the procedure. This directive together with PROC is used to enclose a procedure.

⑪ END

This directive is used after the last statement of the program to inform the assembler that this is the end of a program module.

⑫ ASSUME

It tells the name of a logical segment which is to be used for specified segment.

⑬ STACK

It indicates the start of the stack segment. The default size of stack is 1024 bytes.

(14) • Model:

It provides the info of the segment size that is to be used in a program module. The size of the model may be tiny, small, large, etc.

(15) • Startup:

It indicates the start of the program when using program.

(16) • EXIT:

It indicates the end of the program when using program.

DOS Interrupt Function of 8086:

→ The list of interrupts are:

* INT 10H function:

① Int 10h/AH = 00H → Set video mode input
② Int 10h/ah = 01H → Set text mode cursor shape
③ Int 10h/ah = 02H → Set cursor position
④ Int 10h/ah = 06H → Scroll window up
⑤ Int 10h/ah = 08H → read character and attribute at cursor position.
⑥ Int 10h/ah = 0AH → write character only at cursor position.

* INT 21h functions:

① Int 21h/ah = 01h → read character from standard input, with echo, result is stored in AL
② Int 21h/ah = 02h → write character to standard output. eg: mov ah, 02h (the character must be in dl)
mov dl, 'a'
Int 21h

③ Int 21h/ah = 05h → o/p character to printer

④ Int 21h/ah = 09h → output of a string at DS:DX
String must be terminated by $

⑤ Int 21h/ah = 0Ah → Input of a string to DS:DX
first byte is buffer size, second byte is number of characters actually used.
→ $ not required to terminate the string

⑥ Int 21h/ah = 4Ch → Return control to operating system (Stop program)

WAP in 8086 to add 8-bit number

```
.model small
.stack 100h
.data                                   dB → Define Byte
    val1 dB 25h
    val2 dB 35h
.code
main proc
.startup { mov ax, @DATA
           mov ds, ax
           mov bl, val1
           add bl, val2
           mov result, bl
.exit      { mov ax, 4c00h   or   mov ah, 4ch
             Int 21h
main endp
end main
```

WAP to display alphabets from a to z.

→
```
.model small
.stack 100h
.data
    alpha dB 'a'
.code
  main proc
    .startup
        mov cx,26
        mov bl, alpha
        mov dl, bl
again : mov ah, 02h   } a display.....
        Int 21h
        Inc dl
        loop again
    .exit
  main endp
  end main
```

WAP to display numbers from 0 to 9.

→
```
.model small
space macro
mov dl,' '
mov ah, 02h
int 21h
endm
.stack 100h
.data
  digit dB '0'
.code
  display proc
    .startup
    mov cx,10
    mov bl, digit
again: mov dl, bl
    mov ah,02h
    int 21h
    space
    inc bl
    loop again
    .exit
  display endp
  end display
```


loop

MACRO

It is a group of instructions. The macro assembler generates the code in the program each time where the macro is called. Macro's can be defined by MACRO and ENDM directives.

Example:

```
space  macro ; macro definition
(name)
mov    dl, c   }
mov    ah, 02h } ; body
Int    21h     }
endm ; end macro
```

Macro Vs Procedure

	Procedure	Macro
1	Accessed by CALL and RET instruction during program execution.	Accessed during Assembly with name given to macro
2	Machine code is generated only once in memory	Machine code is generated each time when it is called
3	Less memory (memory efficient)	More memory is required
4	Parameters can be passed in register, memory location, stack.	Parameters passed as a part of the statement which calls macro.
5	long codes	Short codes
6	eg main proc ; main endp end main	eg space macro { endm

WAP to display alphabets 'a to z' with space (Use macro)

→
```
.model small
space macro
mov dl, ' '
mov ah, 02h
Int 21h
endm
.stack 100h
.data
    alpha dB 'a'
.code  display proc
    .startup
    mov cx, 26
    mov bl, alpha
again: mov dl, bl
    mov ah, 02h
    Int 21h
    space
    Inc bl
    loop again
    .exit
    display endp
    end display
```


WAP to display a string 'microprocessor' characterwise in console

→
```
.model small
.stack 100h
.data
    string dB 'microprocessor"
.code
    main proc
    .startup
    mov cx, 14
    mov si, offset string
again: mov dl, [si]
    mov dl, bl
    mov ah, 02h
    Int 21h
    Inc Si
    loop again
    .exit
    main endp
    end main
```

To display a string 'microprocessor' in reverse order characterwise

→
```
.model small
.stack 100h
.data
    string dB 'microprocessor"
.code
    main proc
    .startup
    mov cx, 14
    mov si, offset string
```

```
add si, 13
again : mov bl, [si]
        mov dl, bl
        mov ah, 02h
        int 21h
        Dec si
        loop again
        .exit
        main endp
        end main
```

WAP to transfer a block of data from memory location list1 to list2. Consider the block of six bytes.

→ Title copy of block of data.

```
Dosseg
.model small
.stack 100h
.data
    list1 dB 10,20,30,40,50,60
    list2 dB 6dup(?)
.code
  main proc
    .startup
      mov si, offset list1
      mov di, offset list2
      mov cx, 06
  back: mov al, [Si]
      mov [di], al
      inc Si
      inc di
      loop back
      .exit
      main endp
      end main
```

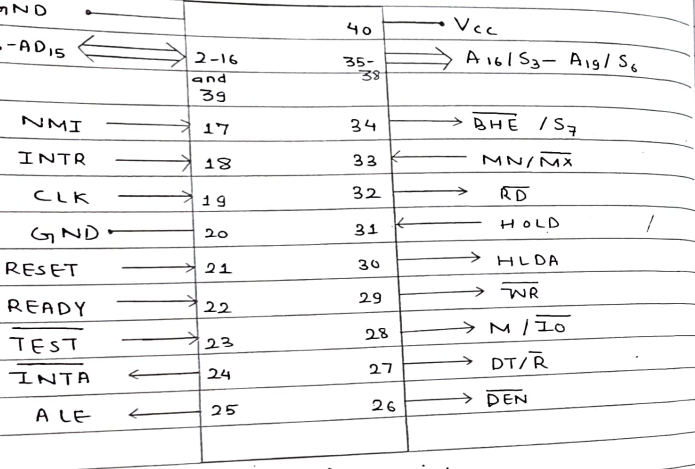

You are given two strings "assembly program" and "language". Now write an assembly language program to display the string 'assembly language program'.

→

```
.model small
 display macro
    mov dx, offset string2
    mov ah, 09h
    int 21h
    end m
.stack 100h
.data
    String1 dB "Assembly Program"
    String2 dB "Language"
.code
  main proc
    .startup
        mov cx, 09
        mov Si, offset string1
back : mov dl, [Si]
        mov ah, 02h
        Int 21h
        Inc Si
        loop back
        display
        mov di, offset string1
        add dl, 10
again : mov dl, [di]
        mov ah, 02h
        Int 21h
        Inc di
        loop again
        .exit
        main endp
```

8086 modes of operation

1. Minimum mode (Single Processor Mode) [MN / $\overline{MX}$ = 1]
2. Maximum mode (Multi Processor Mode) [MN / $\overline{MX}$ = 0]



Starting from 31: For maximum pin

$\overline{RQ}$ / GT0

$\overline{RQ}$ / GT1

$\overline{LOCK}$

$\overline{S_2}$

$\overline{S_1}$

$\overline{S_0}$

Figure: Pin configurations of minimum and maximum mode

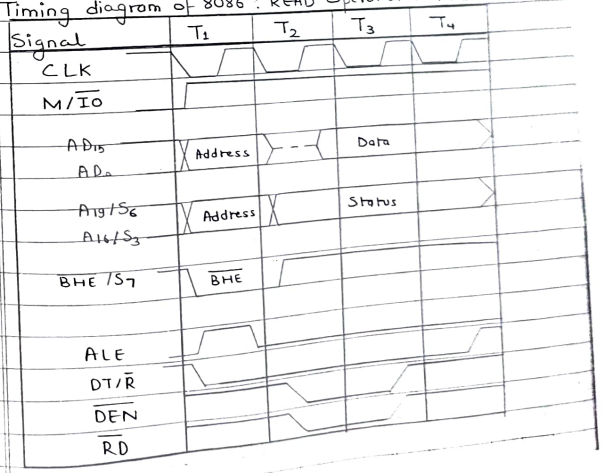
Table:

S_4	S_3	Segment Register
0	0	ES
0	1	SS
1	0	CS
1	1	DS

$\overline{BHE}$	A_0	Word/Byte Accessed
0	0	Whole word
0	1	Upper byte
1	0	Lower byte
1	1	none

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Characteristic
0	0	0	Interrupt Acknowledge
0	0	1	I/O Read
0	1	0	I/O write
0	1	1	Halt
1	0	0	Opcode Fetch
1	0	1	Memory Read
1	1	0	Memory write
1	1	1	Passive.

Timing diagram of 8086 : READ Operation (Minimum Mode)



→ Figure shows the timing diagram for 8086 memory read bus cycle for minimum modes of operation. During T_1 clock of bus cycle, the 8086 sends out the 20-bit memory address on the address bus. As the address lines operates in time multiplexed mode, the address remain on these lines only for clock T_1. For the minimum modes of operation, pin $MN/\overline{MX}$ is kept high.

- The 8086 sends a high signal $M/\overline{IO}$ line to indicate that the data transfer will take place with memory.
- During T_1, an ALE (Address Latch Enable) is sent out so that the address is latched in the external devices.
- During T_2, the AD_0-AD_{15} gives into high impedance state. During T_3 and T_4 data are transmitted on this bus.
- During T_2, T_3 & T_4, the address lines A_{16}/S_3-A_{19}/S_6 carry ~~signals~~ status signals S_3, S_4, S_5 & S_6.
- $\overline{BHE}$ goes low during T_1. It works in conjunction with A_0 that decides whether a byte or word is to be transmitted from/to the addressed memory location
- $\overline{RD}$ goes low during T_2 and the data is read out at T_2 and T_3.
- $\overline{DEN}$ signal is an output enable signal for the octal bus transceivers.
- $DT/\overline{R}$ signal controls the direction of data flow throw transceivers.

For read operation, $DT/\overline{R} \rightarrow 0$

write " , $DT/R \rightarrow 1$

Maximum Mode (Read & Write) → yourself

For Write Operation (Minimum Mode)