Assembler:

- Program that converts assembly language into machine language is called Assembler.
- E.gs. TASM (Turbo Assembler), MASM (Microsoft Assembler), ASM86

Advantages of Assembly Language Programming:

- 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 Hardware 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 comment field which starts 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 is to be loaded into memory.
- It usually converts .exe file to .bin file.
- A Locator program EXE2BIN converts .exe files 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 setting 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 Assembler:

1. One- pass Assembler,
2. Two- pass Assembler

(1) One- pass Assembler:

- This assembler goes through the Assembly Language Program (ALP) once and then translates it into machine code.
- Supports backward reference and problem with forward reference.

E.gs.

Label: MOV AX,CX ..... JMP Label ..... **Remark: Support**	JMP Label1 ..... Label1: MOV AX,CX ..... **Remark: Does not Support**

(2) Two-pass Assembler:

- This assembler scans the assembly language program twice.
- In the first pass , it generates the table of symbols which consists of label with the address assigned to them.
- Forward/Backward reference is supported.
- On the second pass, the assembler translates ALP into machine code.
- E.g. MASM

Macro Assembler:

- Translates the program written in macro language into machine language.
- A macro is a group of instructions that performs the task as a procedure.
- Disadvantage of using procedure are need of stack over head time required to call plus return to calling program.
- Macro sequences execute faster than procedure.
  - No call and return instructions
- Each time encountering macro name, macro assembler replaces it with appropriate instruction sequence (i.e. copy paste)

E.g.

```
MOVE  MACRO A,B ; MOVE is a macro name and A & B are parameters or arguments
PUSH AX
MOV AX,B
MOV A,AX
POP AX
ENDM; End of macro
```

Program:

```
MOVE  VAR1, VAR2  ; MOVE is a macro
```

Note: Macro with/without argument is both possible.

E.g. terminate macro

```
Mov ah, 4Ch
Int 21h
```

Program

At last.

```
....
Terminate
```

ASSEMBLY LANGUAGE PROGRAM FEATURES

#PROGRAM COMMENTS

- The use of Comments throughout a program can improve its clarity, especially in Assembly Language.
- A Comment begins with Semicolon.

EXAMPLE

```
MOV AX, BX      ; Adds the Content of BX with AX and stores the result in AX.
```

#RESERVED WORDS

- Instructions : MOV, ADD
- Directives : END,SEGMENT
- Operators : FAR,OFFSET
- Predefined Symbols : @DATA

#IDENTIFIERS

- An Identifier (or symbol) is a name that you apply to an item in your program that you expect to reference. The two types of identifiers are NAME and LABEL.
- NAME : Refers to the Address of a data item

  COUNTER DB 0
- LABEL: Refer to the Address of an instruction, procedure, or segment.

  MAIN PROC

  A20: MOV AL,BL

#STATEMENTS

- An Assembly Program consists of a set of statements. The two types of statements are:

1. **INSTRUCTION**
   - Instructions such as MOV, ADD, etc which the Assembler translates to Object Code.
2. **DIRECTIVES**
   - Directives tell the Assembler to perform a specific action, such as define a data item etc.

ASSEMBLY LANGUAGE PROGRAMMING USING MASM

GENERAL PATTERN FOR WRITING ALP IN MASM

```
[PAGE DIRECTIVE]
```

[TITLE DIRECTIVE]
[MEMORY MODEL DEFINITION]
[SEGMENT DIRECTIVES]
[PROC DIRECTIVES]
.....
.....
.....
.....
[END DIRECTIVES]

BASIC FORMAT OF ALP BASED UPON THE GENERAL PATTERN

```
PAGE 60,80
TITLE "ALP TO PRINT FACTORIAL NO"

MODEL [MODEL NAME]
STACK
DATA
..... ; INITIALIZE DATA VARIALBLES
CODE

MAIN PROC

.....
.....
.....
..... ; INSTRUCTION SETS
.....
.....
.....
.....
```

MAIN ENDP
END MAIN

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.
- Their main task is to inform the assembler about the start/end of the segment, procedure or program, to reserve the appropriate space for data storage, etc
- They are called pseudo-instructions because they are seem like instructions in program but are not instructions in real and are not converted into object code during assembling.
- The most common Directives are PAGE, TITLE, PROC, END, etc.

- The important assembler directives are explained below:

 # 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

 # 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"

#DOSSEG

- The DOSSEG directive tells the MASM to place the segments(code, data and stack) in the standard order.

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.
- MASM provides simplified Segment Directive.
- 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.

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.

#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

#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

After Using End directive, long comments can also be written later as END will end program execution and the comment won't affect it.

END [PROCEDURE NAME]

#PROCESSOR DIRECTIVE

- Most Assemblers assume that the source program is to run on a basic 8086 level.
- As a result, when you use instructions or features introduced by later processors, you have to notify the assemblers be means of a processor directive as .286,.386,.486 or .586
- This directive may appear before the Code Segment.

#THE EQU DIRECTIVE

- It is used for redefining symbolic names

EXAMPLE

DATAX DB 25

DATA EQU DATAX

#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.

DATA DEFINITION DIRECTIVES

- In assembly language, we define storage for variables using data definition directives.
- Data definition directive create storage during assembling time.
- 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(2 bytes)	DW
DOUBLE WORD (4 bytes)	DD
FAR WORD	DF
QUAD WORD (8 bytes)	DQ
TEN BYTES	DT

For 8086, Only DB and DW used As, 8086 can process either 8 or 16 bit only on maximum.

- **DB**
- The DB directive creates storage for a byte or group of bytes, and optionally assigns starting values.
- e.g. num DB 20h,43h,10h,02h - multiple bytes
  num DB 12h - 1 byte
  data db ? - undefined byte.

- **DW**
- The DW directive creates a storage for a word or list of words
- e.g. data DW 1234h,4567h
  data2 DW abcdh

 # DUP Operator

- The DUP operator allocates multiple occurrence of a value.
- 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

PROCEDURES AND MACROS

Procedures:

- Procedure or a subroutine or a function is a key concept of modular programming, the essential way to conquer complexity.
- A procedure is a reusable set of instructions that has a name.
- Only one copy of procedure is stored in memory; and it can be called as many times as needed.
- "CALL" transfers control to the procedure like a jump; but unlike a jump, procedure has a "RETURN" instruction which returns control to the instruction following the CALL instruction. During the call and return process, the information is stored temporarily in stack.
- Further, nested procedures are also possible. In other words, Procedure A can call Procedure B which in turn calls Procedure C. After completing Procedure C, control returns to Procedure B and after completing Procedure B control returns to Procedure A.
- It is also possible for a procedure to call itself (a recursive procedure).

- A procedure can be in the same code segment as that of the main program (**Intra-segment**). In such a case, we specify only IP as relative distance or indirectly as actual value. This is known as NEAR CALL.
- Moreover, the procedure may also be in 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.
- In a program, procedure starts with PROC directive and ends with ENDP directive.
- **Near CALL Instruction:**

- Similar to Near Jump except that current IP is saved on the stack before transferring control to the new IP with CS remaining the same.
- This instruction is 3-byte long.

Example:

```
Main Program:
|
..
..
CALL MULT ; Calling a procedure, transfer a control to procedure named as MULT.
..
..
; Procedure definition
MULT PROC NEAR USES BX
        MOV AX, 1
        ADD AX, BX
        RET
MULT ENDP
....    ...     ...
CALL SR1
```

- **FAR CALL instruction:**

- FAR CALL is like a FAR JMP in the sense that it can call a procedure available anywhere in the code space.

- In this case, the target address is directly specified as a new CS:IP
- Both current IP and CS are saved on the stack and then control is transferred to the new CS:IP.
- This instruction is 5-byte long.
- Example:

```
SUM1        PROC FAR
..... ...    ..............
SUM1        ENDP
```

Now, a call to SUM1 is assembled as FAR CALL.

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.

Macro:

- Macro is a group of instructions with a name, which provides several mechanisms useful for the development of generic programs.
- When a macro is invoked, the associated set of instructions is inserted in place into the source, replacing the macro name. This "macro expansion" is done by a Macro Preprocessor and it happens before assembly. Thus, the actual Assembler sees the "expanded" source.
- The body of the macro is defined between a pair of directives, MACRO and ENDM.
- Examples of Macro Definitions:

```
; Definition of Macro named SAVE :
SAVE        MACRO
            PUSH AX
            PUSH BX
            PUSH CX
```

```
        ENDM

; Another Macro named RETRIEVE is defined here
RETRIEVE  MACRO
          POP CX
          POP BX
          POP AX
          ENDM
```

Now the macro is called in the main program i.e. macro invocations is as shown below:

```
SAVE
MOV CX, DA1
MOV BX, DA2
ADD AX, BX
RETRIEVE          ; Invoke macro
```

Macro Vs Procedure:

(1) Procedure:

- Only one copy exists in memory. Thus memory consumed is less.
- "Called" when required.
- Return address (IP or CS:IP) is saved on stack before transferring control to the subroutine through CALL instruction. It should be popped 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.

(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. The code is in-place.
- Good if few instructions are in the Macro body.
- No use of stack for operation

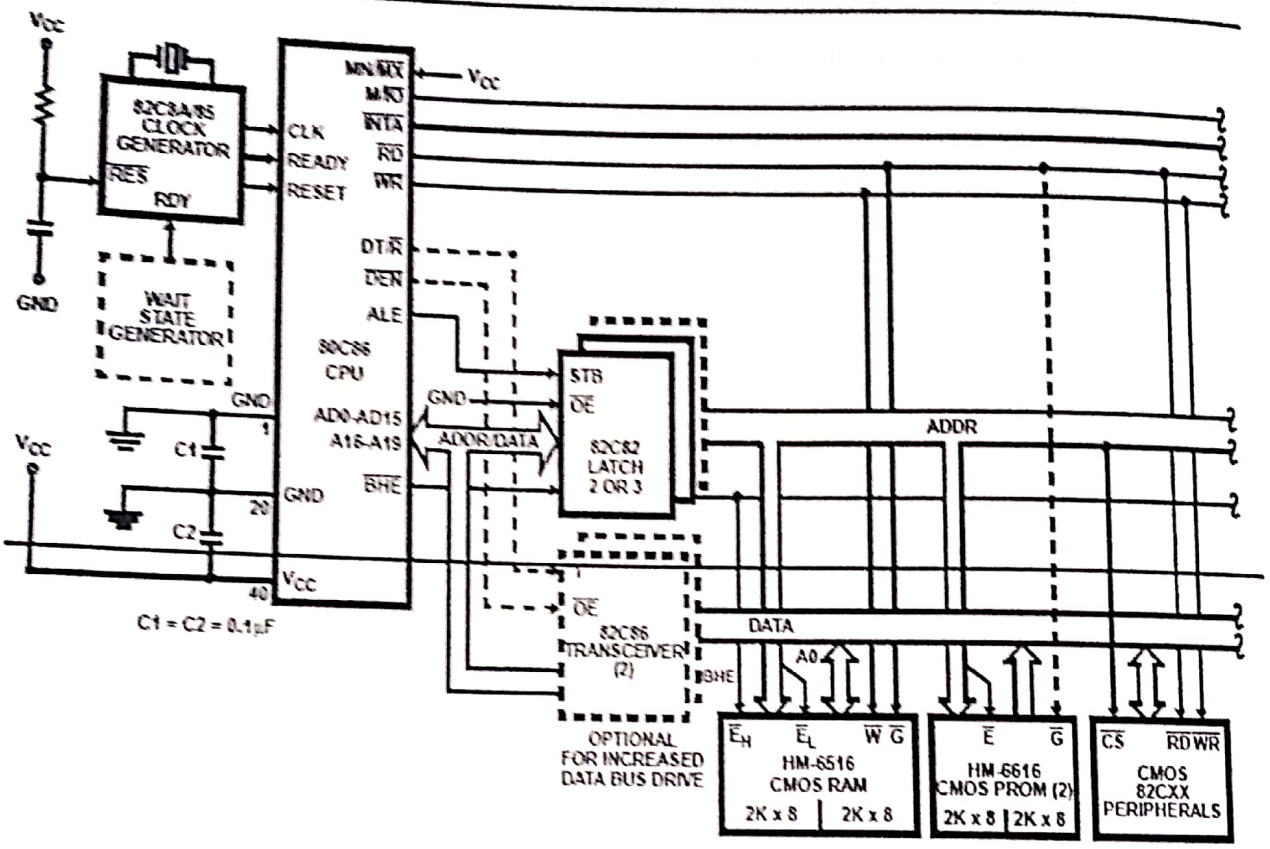
8086 System Timing Diagram

BASIC BUS OPERATION:

- The Three buses of 8086 are Address, Data and Control.
- If Data are written to the memory, the MPU outputs the memory address on the address bus, outputs the data to be written into memory on the data bus, and issues a write( ) to memory.
- If Data are to be read from the memory, the MPU outputs the memory address on the address bus issues a read memory signal and accepts data via data bus.
- 8086 operates in two operating modes. On the basis of the mode it is operating the variation of control signals occurs.

➢ For minimum mode, the control signals are generated by the 8086 processor but in case of maximum mode, the control signals are generated by the Bus Controller as well.

1. MINIMUM MODE CONFIGURATION



➢ 8086 operates in Minimum mode when MN/$\overline{MX}$ = 1,

➢ In this mode only one processor 8086 is used.

➢ The important chips used for operating 8086 in this mode are:

(i) Latches (8282) [Here ALE of 8086 is given to $\overline{STB}$ of 8282]

(ii) Transceiver 8286:

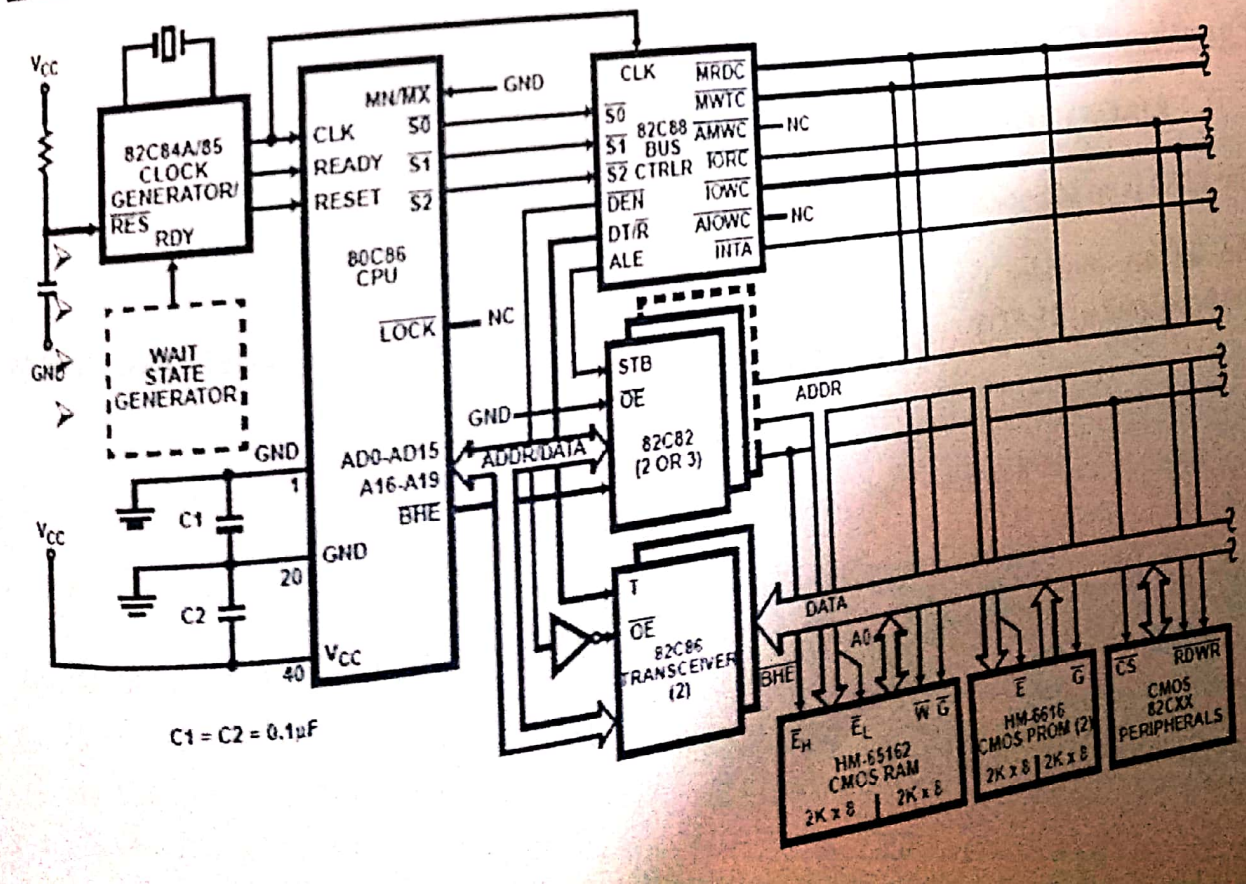
- $\overline{DEN}$ of 8086 is given to $\overline{STB}$ of 8286,
- DT/ $\overline{R}$ of 8086 is given to T of 8286,
- When $\overline{DEN}$ = 0, DT/ $\overline{R}$ =1, then 8086 is transmitting data.

- When $\overline{DEN} = 0$, $DT/\overline{R} = 0$, then 8086 is receiving data.

(iii) Clock Generator 8284:

In minimum modes, 8086 generates control signals such as $\overline{WR}$, $\overline{RD}$, and $M/\overline{IO}$ for external peripheral devices.

2. MAXIMUM MODE CONFIGURATION



- 8086 Works in Maximum mode when MN/$\overline{MX}$ = 0
- In Maximum mode, there is at least one more processor in the system besides 8086
- Clock is provided by 8284 Clock Generator
- The most significant part of the Maximum mode mode circuit is **8288 Bus Controller**
- Instead of 8086, the bus controller provides the various control signals needed to the system in maximum mode
- Address from the address bus is latched into 8282, 8-bit latch, 3 such latches are needed.
- The ALE for this latch is given by 8288 Bus Controller
- The ALE is connected to $\overline{STB}$ of latch
- The data bus is driven through 8286 transceiver, 2 such transceiver are needed
- The transceivers are enabled through DEN signals, while the direction of data is controlled by DT/$\overline{R}$.(DEN for 8288 is active high)
- Both these signals are given by 8288 Bus Controller.

Functional Chips:

1. 8284A: CLOCK GENERATOR

The 8284 is an additional component to the 8086 MPU.

The 8284A provides the following basic functions or signals: Clock generation, RESET synchronization, READY synchronization etc.

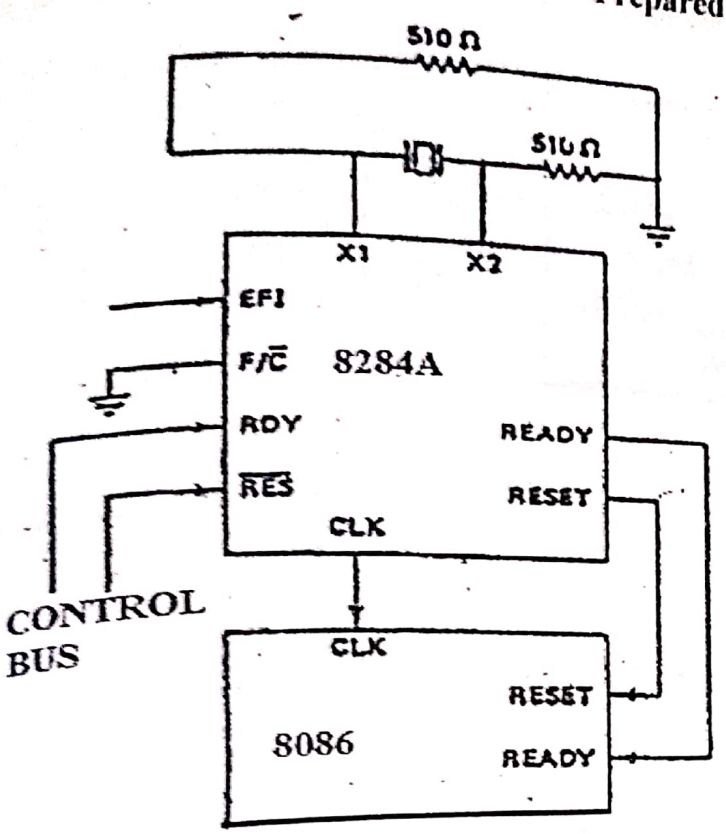


Fig. Connection of 8284A with 8086 microprocessor

8284A can produce clock in two ways:

- By applying output of pulse generator at EFI (External Frequency Input). Now, we have to keep F/ $\overline{C}$ equal to 1.

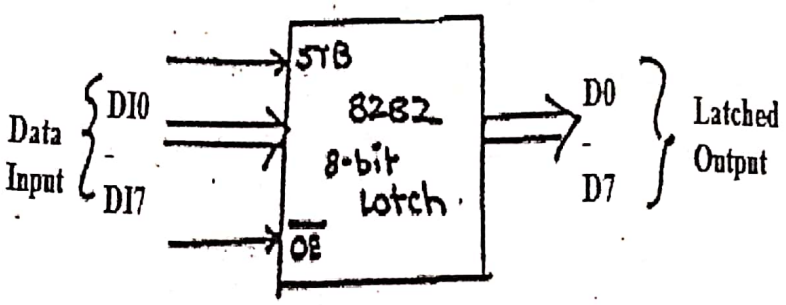
- By connecting crystal oscillator across X_1 and X_2. Now we have to keep F/ $\overline{C}$ (Frequency / Crystal) = 0.

In both the cases, the output clock frequency of 8284 is one-third of input frequency.

2. 8282: ADDRESS LATCH (20 Pin DIP IC)

- In 8086 the address bus is multiplexed with the data bus and status signals. To de-multiplex this bus 8282 address latch is used.
- As the address bus is of 20 bit, three latches are required, each of 8-bit.

The block diagram of 8282 latch is as shown below:



- 8282 address latch is selected, when $\overline{OE}$ is low and whenever $\overline{STB}$ goes high, input is transferred to output. And when $\overline{STB}$ remains low, output remains previous value.

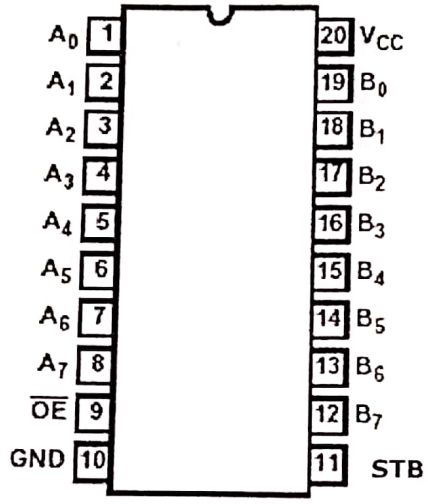


Figure: 8282 Address Latch

PIN OUTS AND FUNCTIONS

#A0-A7

- DATA INPUT Pins.
- Data are inputted to 8082 Address Latch from Data Input Pins.

#B0-B7

- DATA OUTPUT Pins.
- Data are outputted through Data Output pins.
-

#STB

- When STB =0 then it holds Data.
- When STB=1 then data bits are transferred from the Bus.
-

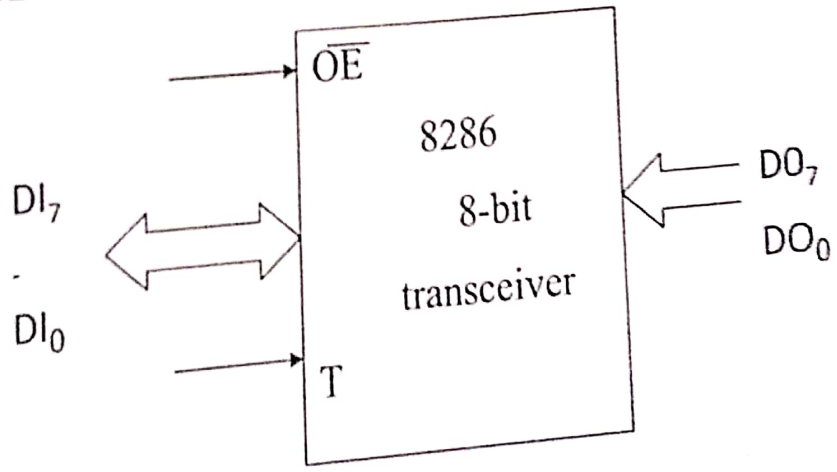
$\overline{OE}$

- Output Enable

8286: TRANSCEIVER

- 8286 Transceiver allows two way communications.
- It acts as bi-directional buffer and increases the driving capability of data bus.

It is enabled when $\overline{OE} = 0$,



T (connected to DT/$\overline{R}$) controls the direction of data:

If T=1, data is transmitted.

If T=0, data is received.

As the data bus of 8086 is of 16 bits, two transceivers are required.

It is available in 20 pin DIP package.

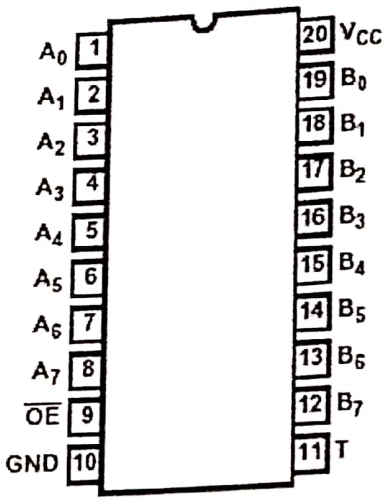


Figure: 8286 Transceiver

PIN OUTS AND FUNCTIONS

#A0-A7

- Local Bus Data I/O Pins.

#B0-B7

- System Bus Data I/O Pins.

$\overline{OE}$ [OUTPUT ENABLE]

- Output Enable.

#T [TRANSMIT CONTROL PIN]

- Transmit Control Pin.
- When T=1, Data are Transmitted.
- Data are inputted from Local Bus Data I/O pins.
- Data are outputted by System Bus Data I/O Pins.
- When T=0,Data are Received.

- Data are inputted by System Bus Data I/O pins.
- Data are outputted by Local Bus Data I/O Pins.

8288 Bus Controller (20 Pin DIP IC):

- It is used to produce different bus control signals like $\overline{MRDC}$ (Memory Read Command), $\overline{MWTC}$ (Memory Write Command), $\overline{IORC}$ (Input Output Read Command), $\overline{IOWC}$ (Input Output Write Command),etc depending on the condition of status signals ($\overline{S2}, \overline{S1}$ and $\overline{S0}$) during Maximum mode operation.

PIN OUTS AND FUNCTIONS

#S2,S1 AND S0

- Status inputs are connected to the status o/p pins on the 8086 MPU.
- These 3 control signals are decoded to generate the timing signals for the system

#CLK [CLOCK]

- The Clock I/P provide internal timing and must be connected to the CLK O/P pin of the 8284A clock generator.

#ALE

- The Address Latch Enable O/P is used to demultiplex address/data bus.

#DEN

- The Data Bus Enable pin control the bi-directional data bus buffer in the system.

#DT/$\overline{R}$

- The Data Transmit/Receive signal is output by 8288 to control the direction of the bi-directional data bus buffers.

#$\overline{AEN}$

- The Address Enable I/P cause the 8288 to enable the memory control signals.

#CEN

- The Control Enable I/P enable the command output pins on the 8288.

#IOB

- The I/O bus mode input selects either the I/O Bus mode or System Bus mode.

#$\overline{AIOWC}$

- The Advanced I/O Write is a command O/P used to provide I/O with an advanced I/O write control signal.

#$\overline{IOWC}$

- The I/O Write command O/P provides I/O with its main write signal.

#$\overline{IORC}$

- The I/O Read command O/P provides I/O with its read control signal.

#$\overline{AMWC}$

- The Advanced Memory Write control pin provides memory with an advanced write signal.

#$\overline{MWTC}$

- The Memory Write Control pin provides memory with its normal write control signal.

#$\overline{MRDC}$

- The Memory Read Control pin provides memory with a read control signal.

#$\overline{INTA}$

- The Interrupt Acknowledge O/P acknowledges an Interrupt Request input applied to INTR Pin.

#MCE/$\overline{PDEN}$

- The Master Cascade / Peripheral Data Output select cascade operation for an interrupt controller if IOB is grounded.
- It enables the I/O Bus Transceiver if IOB is tied high.

Bus Cycle For Minimum Mode (Memory Read or I/O Read):

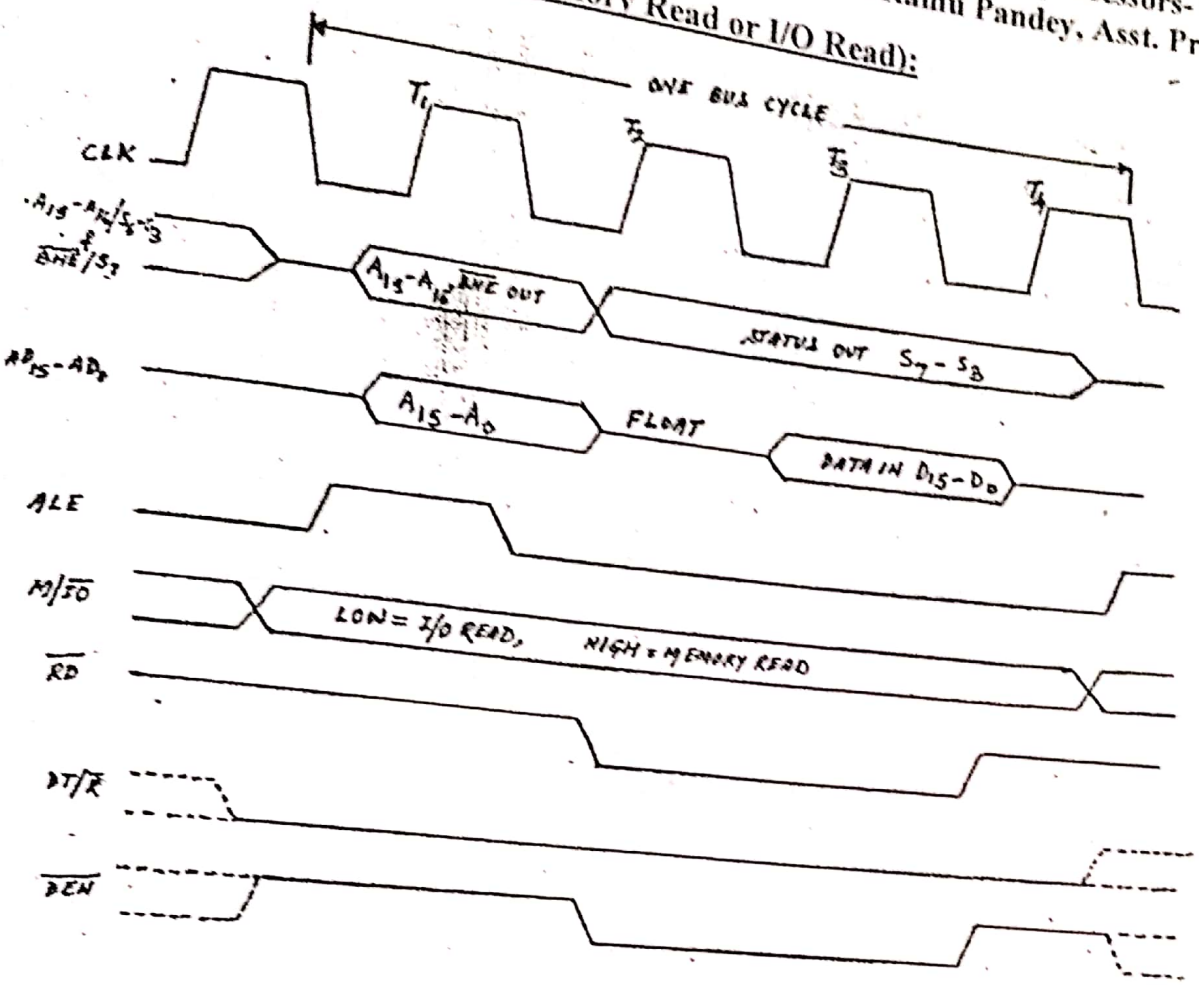


Fig. Bus Cycle For Minimum mode (I/O Read or Memory Read)

During T1 state:

- When ALE goes high, o/p of latch will be $\overline{BHE}$ and A19 – A0 (A19- A16 will be zero for I/O operation)
- DT/$\overline{R}$ go low and thereby 8086 inform 8286 transceiver to receive data from i/p device or memory.
- M/$\overline{IO}$ takes proper value, '0' for I/O operation and '1' for memory operation.

During T2 state:

- The status signals S7- S3 available on status bus.
- $\overline{RD}$ goes low and so memory or i/p device enabled to give data.
- $\overline{DEN}$ goes low and data buffer of 8286 is enabled.

Note: If the peripheral device ready for data transfer, then ready signal goes high. If they are not ready for data transfer, then ready signal goes low. READY signal has to be given before T3 or at the beginning of T3.

During T3 state:

- Microprocessor check ready signal. If it is high microprocessor enters T4 state, if it is low microprocessor produces WAIT state and then checks READY.

  If it is high, it enters into T4. If it is low, it repeats the above process.

During T4 state:

- Microprocessor accepts data from data bus (data bus already received data from memory or input device)
- $\overline{RD}$ goes high and so memory or input device is disabled.
- $\overline{DEN}$ goes high and so data buffer of 8286 is disabled.

BUS CYCLE FOR O/P OPERATION (Memory Write or I/O Write)

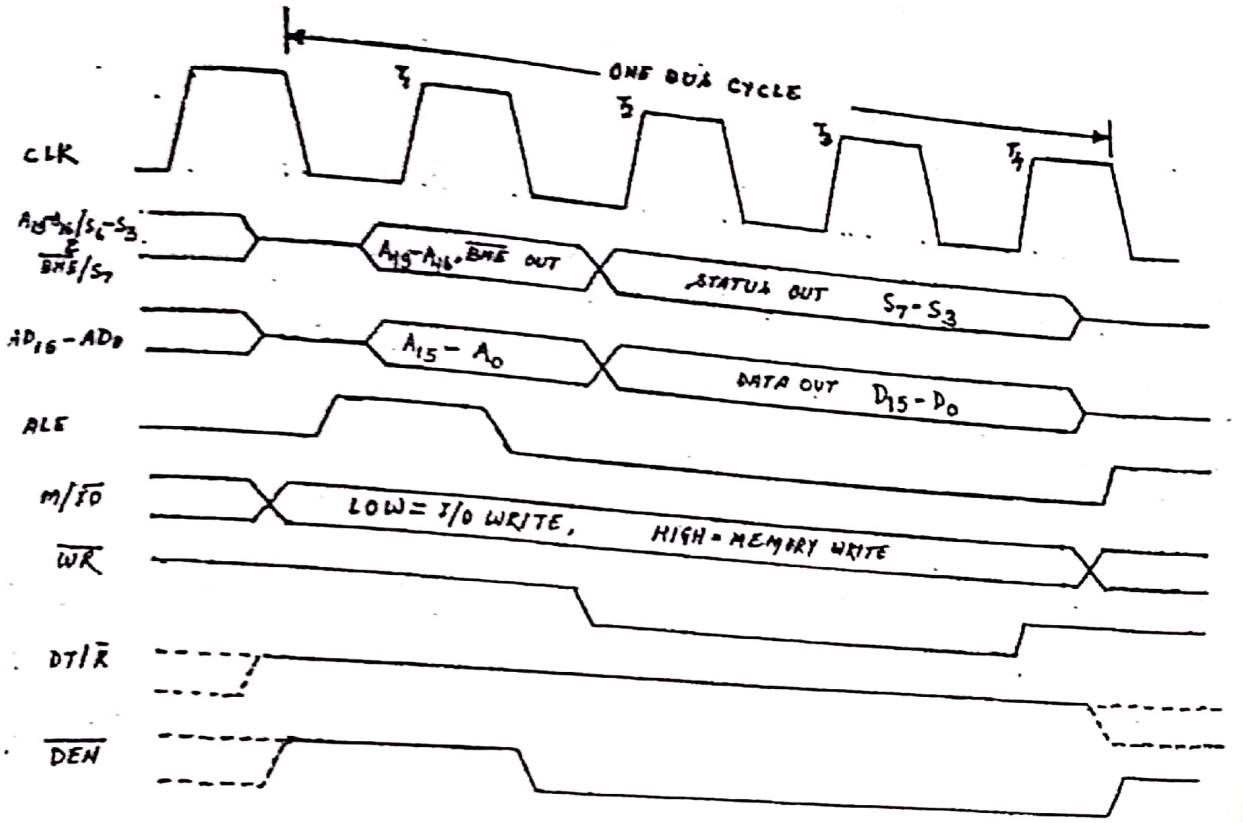


Fig. Bus Cycle For O/P Operation (Memory Write or I/O Write)

During T1 state:

- When ALE goes high, o/p of latch will be $\overline{BHE}$ and A19 – A0 (A19- A16 will be zero for I/O operation)
- DT/$\overline{R}$ goes low and thereby 8086 informs 8286 transceiver to receive data from i/p device or memory.
- M/$\overline{IO}$ takes proper value, '0' for I/O operation and '1' for memory operation.

During T2 state:

- The status signals S7- S3 available on status bus.
- $\overline{WR}$ goes low and so memory or o/p device enabled to give data.
- $\overline{DEN}$ goes low and data buffer of 8286 is enabled.

During T3 state:

- Microprocessor check ready signal. If it is high microprocessor enters T4 state, if it is low microprocessor produces WAIT state and then checks READY.

  If it is high, it enters into T4. If it is low, it repeats the above process.

During T4 state:

- Microprocessor accepts data from data bus (data bus already received data from memory or input device)
- $\overline{WR}$ goes high and so memory or output device is disabled.
- $\overline{DEN}$ goes high and so data buffer of 8286 is disabled.

8086 System Timing Diagram for Maximum Mode:

- In Maximum mode, $\overline{S2}$, $\overline{S1}$ and $\overline{S0}$ are given to Bus controller 8288. 8288 Bus Controller is used to produce different control signals depending on the combination of those signals $\overline{S2}$, $\overline{S1}$ and $\overline{S0}$. The generation of control signals is as shown below:

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

$\overline{MRDC}$ **(Memory Read):** It is used to read data from memory.

$\overline{MWTC}$ **(Memory Write):** It is used to write data from memory.

$\overline{IORC}$ **(I/O Read):** It is used to read data from I/O device.

$\overline{IOWC}$ **(I/O Read):** It is used to write data into I/O device.

$\overline{AMWTC}$ **(Advanced Memory Write Command):** It is similar to $\overline{MWTC}$ except one difference i.e. it is activated one clock cycle earlier. This gives slow memory, an extra clock cycle to prepare itself to accept data.

$\overline{AIOWC}$ **(Advanced IO Write Command):** It is similar to $\overline{IOWC}$ except one difference i.e. it is activated one clock cycle earlier. This gives slow I/O device, an extra clock cycle to prepare itself to accept data.

ALE (Address Latch Enable): It is used to enable the address latch.

DT/$\overline{R}$ (Data Transmit/ Receive): It is given to transceiver. It is used to control direction of dataflow.

DEN (Data Enable): It is given to transceiver through a NOT gate. It is used to enable data buffer of transceiver.

$\overline{AEN}$, IOB, CEN: They are input signal. They are used in multi-processor environment. 8288 is enabled when **CEN** is high. If IOB=0, $\overline{AEN}$ =X, then 8288 is in I/O bus mode. When CEN=0, 8288 is disabled.

MCE/$\overline{PDEN}$: It is o/p signal. In system bus mode, it is known as MCE (Master Cascade Enable). It is used to control cascaded 8259.

Bus Cycle for Input Operation (I/O Read or Memory Read) Maximum Mode:

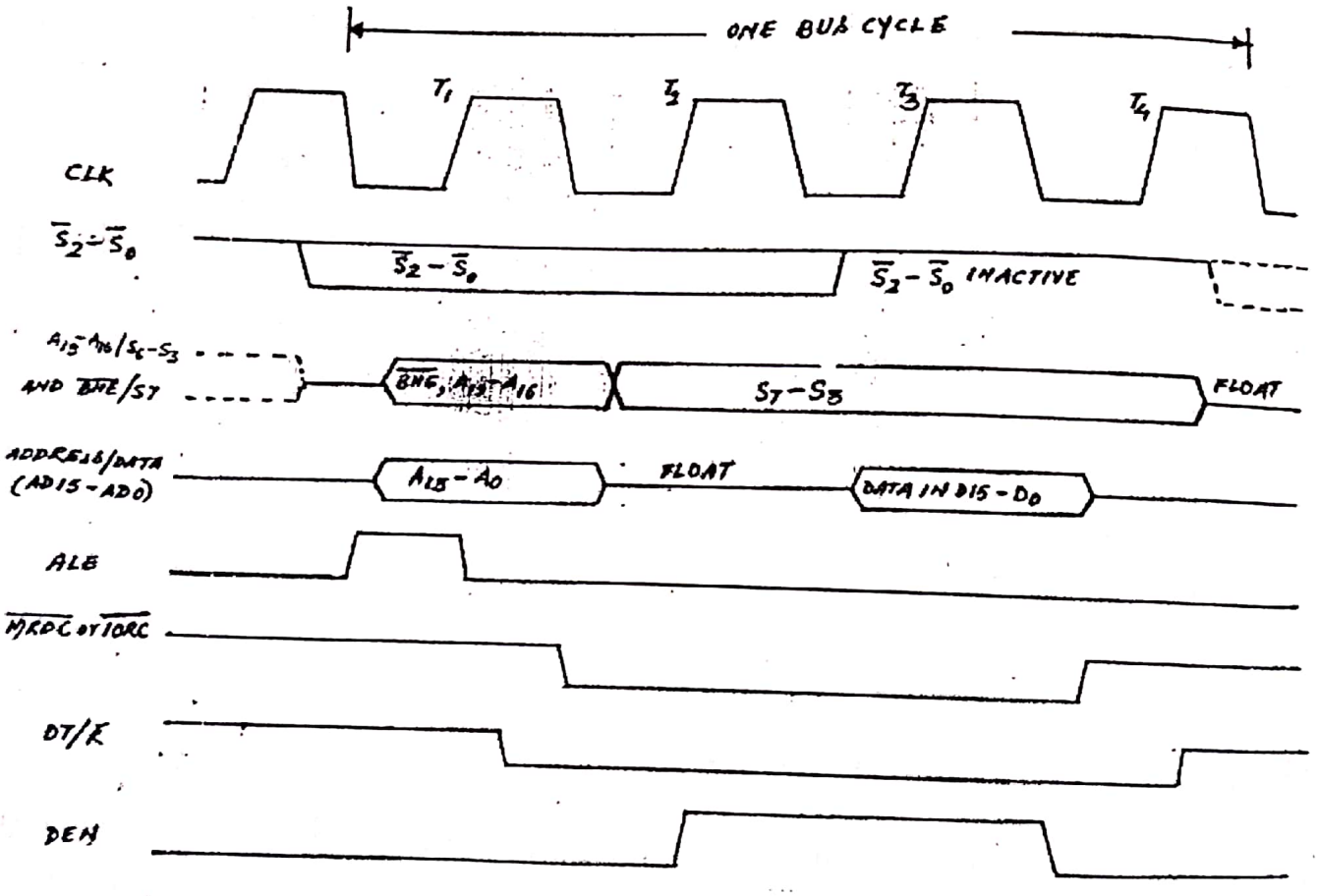


Fig. 8086 Maximum mode for Input Operation (I/O Read or Memory Read)

During T1 State:

- At the beginning of T1, $\overline{S2}$, $\overline{S1}$ and $\overline{S0}$ takes proper value; these values are given to bus controller 8288. Now 8288 produce proper control signals.

- First it makes ALE high and so o/p of latch will be $\overline{BHE}$ and A19-A0
- Then it makes DT/$\overline{R}$ low so transceiver can receive data.

During T2 State:

- The status signal S7- S3 is available on status bus.
- DEN goes high and due to not gate its o/p will be low. Data buffer of transceiver is enabled.
- $\overline{MRDC}$ or $\overline{IORC}$ goes low and so memory or i/p device is enabled to give data on data bus

During T3 State:

- Microprocessor check READY signal, if it is high microprocessor enters into T4 state otherwise waits for it to become high.

During T4 State:

- Microprocessor accepts data from data bus.
- $\overline{MRDC}$ or $\overline{IORC}$ goes high so memory is disabled.
- DEN goes low and due to not gate its o/p goes high so data buffer of transceiver is disabled.

Bus Cycle for Output Operation (I/O Write or Memory Write) Maximum Mode:

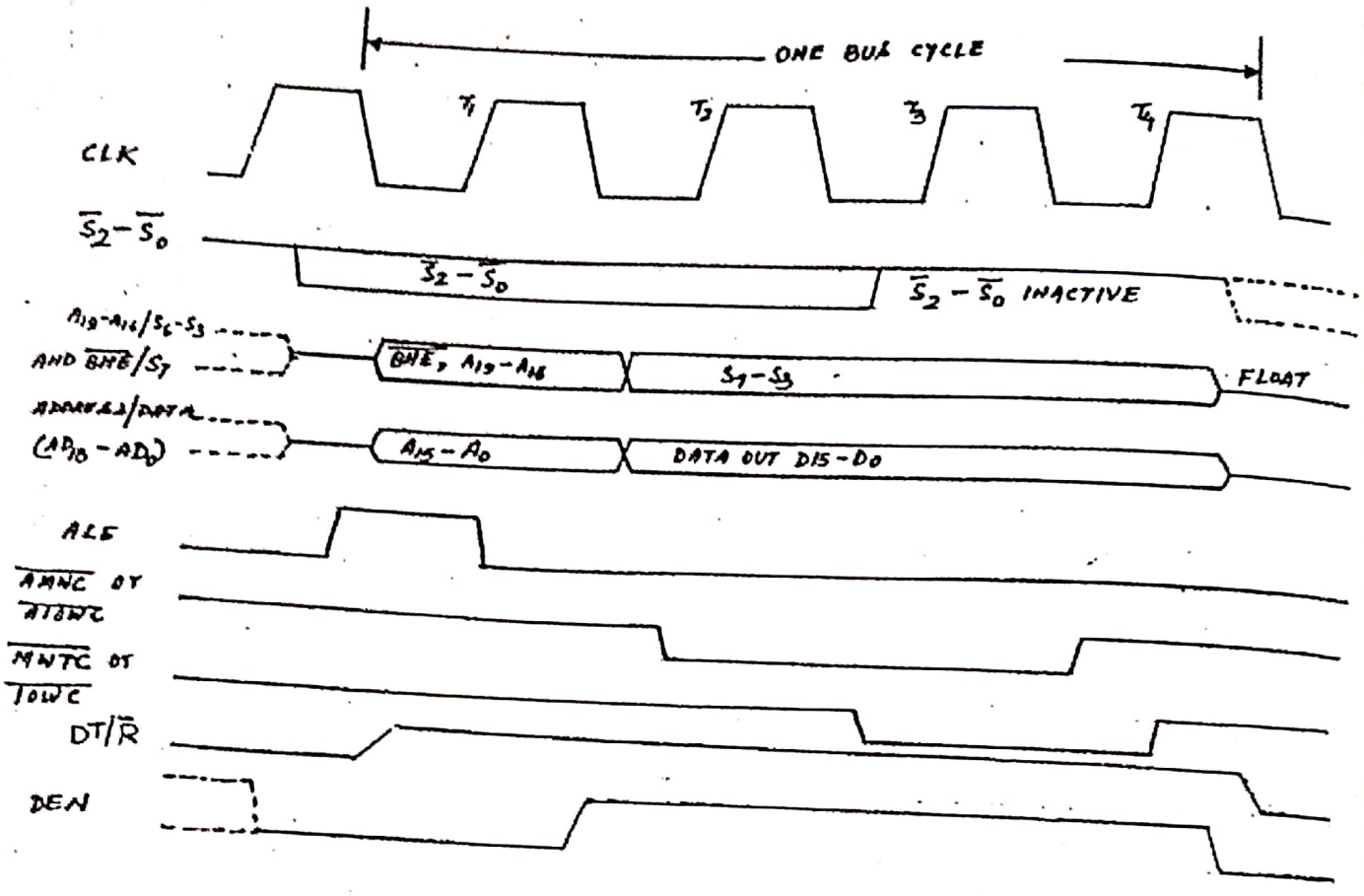


Fig. 8086 Maximum mode for Input Operation (I/O Write or Memory Write)

During T1 State:

- At the beginning of T1 (or just before T1), $\overline{S2}$, $\overline{S1}$ and $\overline{S0}$ takes proper value; these values are given to bus controller 8288. Now 8288 produce proper control signals.
- First it makes ALE high and so o/p of latch will be $\overline{BHE}$ and A19-A0
- Then it makes DT/$\overline{R}$ goes high so transceiver can transmit data.
- DEN goes high and due to NOT gate its o/p goes low and so data buffer of transceiver enabled.

During T2 State:

- The status signal S7- S3 is available on status bus.

- $\overline{AMWTC}$ or $\overline{AIOWC}$ goes low and so memory or i/p device is enabled to give data on data bus

During T3 State:

- Microprocessor check READY signal, if it is high microprocessor enters into T4 state otherwise **waits** for it to become high.

During T4 State:

- Microprocessor accepts data from data bus.
- $\overline{AMWTC}$ or $\overline{AIOWC}$ goes high so memory or I/O device is disabled.
- DEN goes low and due to not gate its o/p goes high so data buffer of transceiver is disabled.

Introduction to Intel 80386:

- Intel 80386, also known as i386 or 386 is a 32-bit processor introduced in 1985.
- As the original implementation of the 32-bit extension of the 80286 architecture, the 80386 instruction set, programming model, and binary encodings are still the common denominator for all 32-bit x86 processors, which is termed as ***i386-architecture***, *x86* or *IA-32*.
- It has 32-bit address bus.
- The 32-bit ALU allows the 80386 to process data faster and the 32-bit address bus allows the 80386 to address up to 4GB of memory.
- The 80386 featured three operating modes: real mode, protected mode and virtual mode.
- The protected mode, which was present in 80286 was extended to allow the 386 to address up to 4GB, so the memory segments can be as large as 4GB.
- The memory management circuitry and protection circuitry in 80386 are improved over that in 80286, so 80386 is much more versatile as a CPU in a multiuser system.
- The 80386 has a "Virtual 8086" mode, which allows it to easily switch back and forth between 80386 Protected-mode tasks and 8086 real mode tasks.
- The 80386 processor is available in two different versions:
  - the 386 DX, and
  - the 386 SX
- The 386 DX has a 32-bit address bus and 32-bit data bus. It is packaged in the 132 pins ceramic pin grid array package.
- The 386 SX, which is packaged in the 100-pin flat has the same internal architecture as 386 DX, but it has only 24-bit address bus and 16-bit data bus.
- The lower cost of packaging and ease of interfacing to 8-bit and 16-bit memory and peripherals make 386 SX suitable for use in lower cost system. But address range and memory transfer rate are lower than 386DX.

Microprocessor 8086 Programs:

Notes:

(i). All programs are tested in 'masm611' and verified. (The 8086 emulator (emu8086) does not support some assembler directives like –dosseg(the optional directive) so the programmer can omit it to execute the program in emu8086)

(ii). '.startup' directive works similar to as mov ax,@data , mov ds,ax i.e. initialization of data segment and ' .exit' or 'end' directive works similar to 'end proc' and 'end main' functions.

1.

Title program to read a character from keyboard and display it on screen

```
dosseg
.model small
.stack
.data
.code
  main proc    ; start procedure with procedure name main
  mov ax,@data       ;  initialize data segment
  mov ds,ax
  mov ah,01h ; accepts the character from keyboard & store in 'al' register & echoes
  int 21h
 mov dl, al    ; transfers the input character from 'al' to 'dl' register to display it
 mov ah, 02h ; displays the character in 'dl' register on screen
 int 21h
mov ah,4ch  ; terminates the program
int 21h
```