CHAPTER – 4

16-bit Microprocessor and Programming (13hrs)

Intel 8086 Microprocessor:

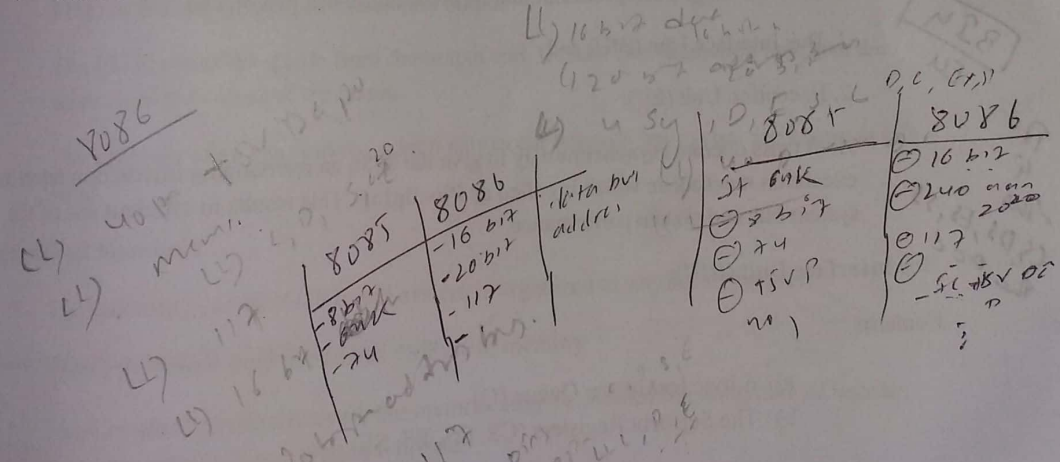
Features:

- Released in 1978 by Intel Corporation.
- It has **16-bit data bus** i.e. it can access 16-bit data in one operation. It's ALU and internal data registers are also 16-bit; hence 8086 is a **16-bit µp**.
- 8086 has a **20 bit address bus** and can access up to 2^{20} memory locations[00000H - FFFFFH] (1 MB)[16 times more than 8085].

- 8086 supports **Pipelining**
  - Fetching the next instruction while executing the current instruction is called Pipelining. 8086 prefetches up to 6 instruction bytes from memory and queues them in order to speed up instruction execution. Pipelining highly improves the performance of the system.

- 8086 supports **Memory Segmentation**:
  - Segmentation means dividing the memory into logical components. Here, the memory is divided into 4 segments: **Code, Data, Stack and Extra Segments.**

- 40-pin IC with +5V DC power supply.
- Available in 3 versions- 8086(Clock rate 5MHz), 8086-2(8 MHz) & 8086-1(10 MHz).
- 8086 is designed to operate in two modes, Minimum and Maximum.
- Have 117 different instructions.



Internal Architecture of 8086 Microprocessor:

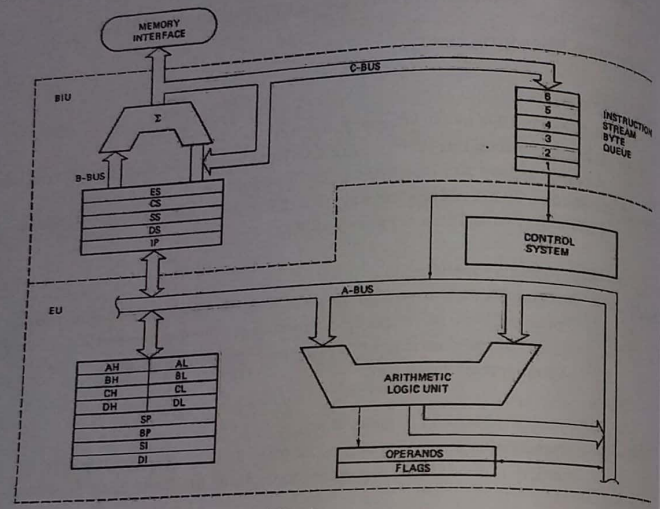


FIGURE 8086 internal block diagram. (*Intel Corp.*)

Explanation:

- 8086 is a 2-stage microprocessor and its architecture has two blocks:

  1. Bus Interface Unit (BIU) and

  2. Execution Unit (EU).

- Both units operate **asynchronously** to give the 8086 an overlapping instruction fetch and execution mechanism which is called as **Pipelining**. This results in efficient use of the system bus and system performance.

Bus Interface Unit (BIU):

Contains

a) 6-byte Instruction Queue (Q)
b) The Segment Registers (CS, DS, ES, SS).
c) The Instruction Pointer (IP).
d) The Address Summing block (Σ)

- BIU is the interface of 8086 to the outside world.
- It is responsible for all external bus operations.
- It performs the following functions:
  - It generates 20-bit **physical address** for memory access.
  - **Fetches instructions** from memory.
  - **Transfers data** to and from the memory and IO.
  - Supports **pipelining** using the 6-byte instruction queue.

THE QUEUE (Q)

- The BIU uses a mechanism known as an **instruction stream queue** to implement pipeline ***architecture.***
- This queue permits pre-fetch of up to **6 bytes** of instruction code. Whenever the queue of the BIU is not full, it has room for at least two more bytes and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by pre-fetching the next sequential instruction.
- These pre-fetching instructions are held in its FIFO queue. With its 16 bit data bus, the BIU fetches two instruction bytes in a single memory cycle.
- After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output.
- The EU accesses the queue from the output end. It reads one instruction byte after the other from the output of the queue.
- The intervals of no bus activity, which may occur between bus cycles, are known as *idle state.*

Segmented Memory:

- The memory in an 8086/88 based system is organized as segmented memory.
- The CPU 8086 is able to address 1Mbyte of memory.
- The Complete physically available memory may be divided into a number of logical segments with each segment of 64kbyte.

- A segment is an area that begins at any location which is divisible by 16

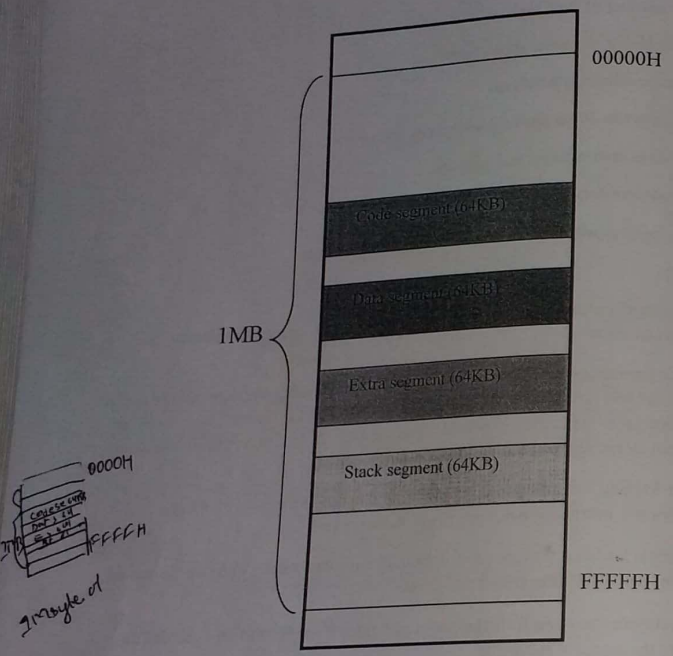


Fig. Physical Memory

- Each of these segments can be used for a specific function.
  - Code segment is used for storing the instructions.
  - The stack segment is used as a stack and it is used to store the return addresses.
  - The data and extra segments are used for storing data byte.

- The 16 bit contents of the segment registers in the BIU actually point to the starting location of a particular segment.

Note: Segments may be overlapped or non-overlapped (for small programs which do not need all 64 KB in each segment can overlap.)

Segment Registers:

CS (Code Segment):

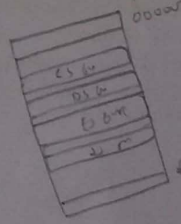
- This segment is used to hold the **program** to be executed.
- Instruction fetch operation is performed in the **CS** memory.
- Holds upper 16 bit of starting address i.e. base address for the code.
- BIU always inserts zeros for the lowest 4 bits of 20 bit starting address.
- Instruction Pointer (IP) holds offset for CS.

SS (Stack Segment):

- Upper 16 bit of starting address for stack memory is kept in this register.
- Stack Pointer (SP) in EU holds the 16 bit offset address of the **Top of the Stack**.
- Base Pointer (BP) holds the 16-bit offset address during **random access**.
- So physical address given by SS and SP.

DS (Data Segment):

- Where data and operands are stored.
- This segment also holds the **source** operands during string instructions.
- DS register points current data segment.
- Operands for most instructions are fetched from this segment.
- The 16 bit content of SI (Source Index) or DI (Destination Index) is used as offset for computing 20 bit physical address.



ES (Extra Segment):

- Data in access of 64KB during some string instructions use ES to hold the destination data.
- ES and DI are used to determine 20bit physical address.

Summary:

- CS : IP,
- SS : SP or BP,
- DS : DI or SI,
- ES : DI ,for string instructions

Advantages of Segmentation:

- It permits the programmer to access 1MB using only 16-bit address.
- It divides the memory logically to store the Data, Instructions and Stack separately.
- Segmentation is very useful for multi-user environment.

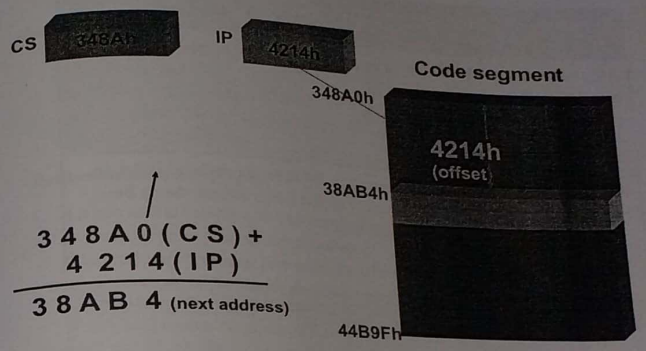
Disadvantages of Segmentation:

- Although the total memory, 16 * 64 KB , at a time only 4 * 64 KB memory can be accessed,
- Any memory locations need to be expressed using 2 registers (Base and Offset).

Instruction pointer & summing block

- The instruction pointer register contains a 16-bit offset address of instruction that is to be executed next.
- The IP always references the Code segment register (CS).
- The value contained in the instruction pointer is called as an **offset** because this value must be added to the base address of the **code segment**, which is available in the CS register to find the **20-bit physical address**.
- The value of the instruction pointer is incremented after executing every instruction.
- To form a 20bit address of the next instruction, the 16 bit address of the IP is added (by the address summing block) to the address contained in the CS, which has been shifted four bits to the left.

The following example shows the CS: IP scheme of address formation:



Execution Unit (EU):

- It **fetches** the instructions from the **Instruction Queue, decodes and executes it.**
- It performs **arithmetic**, **logic**, **decision-making** and **data transfer** operations.
- It sends requests signals to the BIU to access the external module.
- Generates control signals,

The main components of EU are as follows:

a) **General Purpose Registers :**
- 8086 has four 16-bit general purpose registers **AX, BX, CX** and **DX.**
- These are available to the programmer for arithmetic, logic and data transfer operations.
- Each of these registers can be divided into two 8-bit registers like- **AH-AL, BH-BL,CH-CL and DH-DL**

16 bit reg AX, BX, CX, DX

8 bit : AH-AL
BH-BL
CH-CL
DH-DL

Apart from their general use, these registers also have some specific functions as follows:

Register	Purpose
AX(16 bit accumulator)	Used for I/O instruetions , multiplication & division instructions also use AX or AL.
BX(Base Register)	Stores address information. BH & BL are similar to H & L in 8085.Holds the memory address (offset address) during Base Relative Addressing Mode.
CX(Counter Register)	Some instructions like- shift, rotate, loop and string instructions use contents of CX as counter. Decrement CX automatically by 1 without affecting flags.
DX(Data Register)	Used to hold high 16 bit result (data) in 16 X 16 bit multiplication, or 16 bit dividend (data) before 32 /16 bit division or 16 bit remainder after division. Also used to hold the address of I/O port in Indirect I/O Addressing Mode.

b) *Special Purpose Registers: Pointers & Index Registers*

- Used to keep offset addresses.
- Used in various forms of memory addressing.
- The different Pointers and Index Registers are as follows:

Stack Pointer (SP) [16-bits]:

- Holds the **offset address** of the top most elements of the stack in the Stack Segment.
- Used with the SS Register to calculate the physical address in the Stack Segment.
- It is used by Instructions like- PUSH, POP, CALL, RET, etc.

Base Pointer (BP) [16-bits]:

- It is used to hold the **offset address** of the stack in **Random Access Mode.**

- It can be used with SS Register to access the stack randomly.

Source Index (SI) [16-bits]:

- Normally used to hold the offset address for the data segment but can be used for the Stack Segment.
- Is required for some string operations.
- When string operations are performed, the SI register points to memory locations in the data segment which is addressed by the DS register. Thus, SI is associated with the DS in string operations.

Destination Index (DI) [16-bits]:

- It is also used to hold the offset address for the data segment or the stack segment.
- Is also required for some string operations.
- When string operations are performed, the DI register points to memory locations in the data segment which is addressed by the ES register. Thus, DI is associated with the ES in string operations.

Note: SI and DI registers may also be used to access data stored in arrays.

c) Arithmetic and Logic Unit (ALU) [16-bits]:

- It has a 16-bit ALU. It performs 8/16 bit binary arithmetic and logic operations.

d) Operand Register:

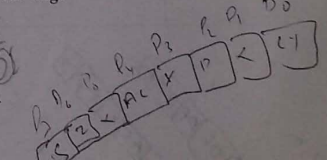
- It is a 16-bit register used by control register to hold the operands temporarily.
- It is not available to the Programmer.

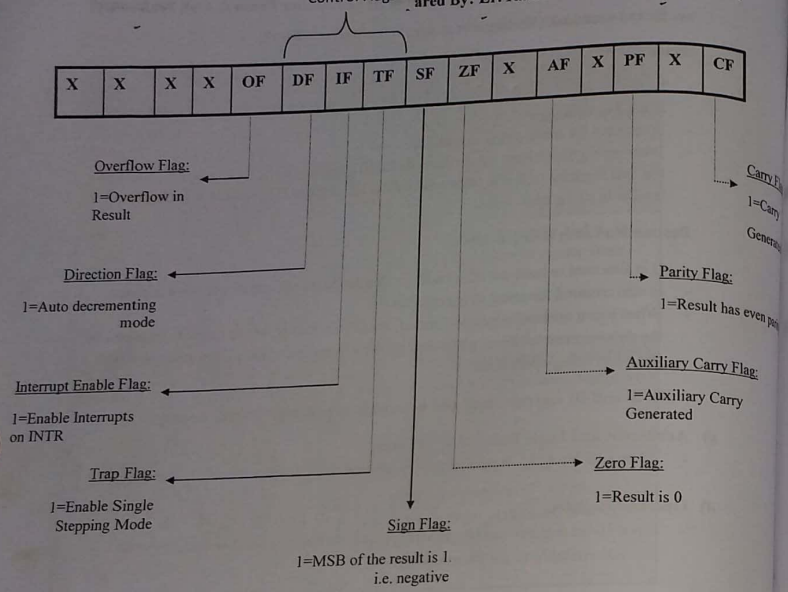
e) Instruction Register and Instruction Decoder:

- The EU fetches an opcode from the queue into the **Instruction Register.**
- The **Instruction Decoder** decodes it and sends the information to the control circuit for execution.

f) Flag Register (16-bits):

- A flag is a flip flop which indicates some conditions produced by the execution of an instruction or controls certain operations of the EU.
- In 8086 The EU contains
  - a 16 bit flag register
  - 9 of the 16 are **active flags** and remaining **7 are undefined**.
    - **6 flags** indicate some conditions- **status flags**
    - **3 flags – control flags**





Flag	Purpose
Carry (CF)	**Status Flag** Set whenever there is a carry (or borrow) out of MSB of a result (D7 bit for an 8-bit operation and D15 bit for a 16-bit Operation)
Parity (PF)	**Status Flag** PF=0; odd parity, PF=1; even parity.

Auxiliary Carry (AF)	**Status Flag** Holds the carry (half – carry) after addition or borrow after subtraction between bit positions 3 and 4 of the result (For example, in BCD addition or subtraction.)
Zero (ZF)	**Status Flag** Shows the result of the arithmetic or logic operation. Z=1; result is zero. Z=0; The result is not 0
Sign (SF)	**Status Flag** Holds the sign of the result after an arithmetic/logic instruction and set if MSB of the result is 1.Execution. S=1; negative, S=0;positive
Trap (TF)	**Control flag.** Setting TF puts 8086 in single step mode, where 8086 generates internal interrupt after execution of each instruction.(1=Trap on; 0= trap off)
Interrupt (IF)	**Control flag.** Setting IF causes 8086 to recognize external mask able interrupt (e.g INTR), clearing IF disables this interrupt.
Direction (DF)	**Control flag.** Used by string manipulation instructions. Setting DF causes string instruction (i.e. SI & DI) to auto decrement. Clearing DF causes string instruction to auto increment.
Overflow (OF)	**Status Flag** Overflow occurs when signed numbers are added or subtracted. An overflow indicates the result has exceeded the capacity of the destination location. Checked by INTO (Interrupt Overflow instruction)

Addressing Modes of 8086:

- Way of defining operands.
- Way in which processor can access data.

Types:

(i) Immediate Addressing Mode
(ii) Direct Addressing Mode
(iii) Register Addressing Mode
(iv) Register Indirect Addressing Mode
(v) Register Relative Addressing Mode
(vi) Base + Index Addressing Mode, &
(vii) Base Relative + Index Addressing Mode.

(i) Immediate Addressing Mode:

-operand is specified in the instruction itself.

E.gs. MOV AL, 39H

ADD AX, 4836H

Note: MVI is not used in 8086.

(ii)Direct Addressing Mode:

-operand offset is given in the instruction as a 16 bit displacement.

e.gs. (i). ADD AL, [0301H]; Adds the contents of offset address 0301H to AL.

(Let the content of DS = 4000H then content of (40000H + 0301H =) 40301H is added to the content of AL and result is stored in AL)

(ii) . ADD [AEB2H], AX; Adds the contents of AX to the content of memory location AEB2 H and AEB3 H. (i.e. [AEB3H] + AH and [AEB2H] + AL, result is stored in AEB3H and AEB2H) .

(iii) Register Addressing Mode:

-operand is placed in one of the 16/8 bit general purpose register. (*Registers must be of same size.*)

E.g. MOV AL, BL

MOV AX, CX; Content of CX is transferred to Accumulator.

MOV SI, DI ; Content of DI is copied to SI register.

(iv) Register Indirect Addressing Mode:

-the operand offset is placed in one of the register BX (*not CX, DX*), BP, SI, DI as specified in the instruction.

e.g. (i). MOV AX,[BX] ; memory address= content of BX

If, [BX] = 0301H and

[0301H] = 53H

[0302H] = 95H, then

[AX] = 9553H.

(ii). ADD AL, [SI] ; content of memory location addressed by SI is added to content of AL and result placed in AL.

(v.) Register Relative Addressing Mode:

-operands offset is the sum of an 8-bit or 16-bit displacement and the content of base register BX or index register SI.

e.g. MOV AX,[BX + 05H] : if BX=0301H, offset= 0301H + 05H = 0306H.

MOV [SI + 20H], BL.

(vi) Base + Index Addressing Mode:

- Uses base registers (BP or BX) and one index registers (SI or DI) to indirectly address memory.

- Base register = beginning location of memory.

Index register = relative position of an element in an array.

so, offset = [BX or BP] + [SI or DI]

e.g. (i) ADD AX, [BX + SI];

(ii) MOV BL, [BP + SI]; if PA = 4000h, [BX]= 1250h, [SI] = 8214h then, actual address = 40000+1250+8214 = 49464h, so BL [49464h], after execution of this instruction.

(vii) Base Relative + Index Addressing Mode:

- combines the features of 'Base + index' and 'register relative' addressing modes.

Here, operand offset is given by,

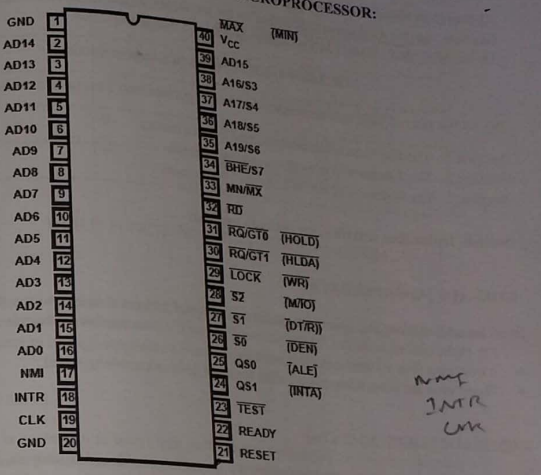
Offset = [BX or BP] + [SI or DI] + 8/16 bit displacement.

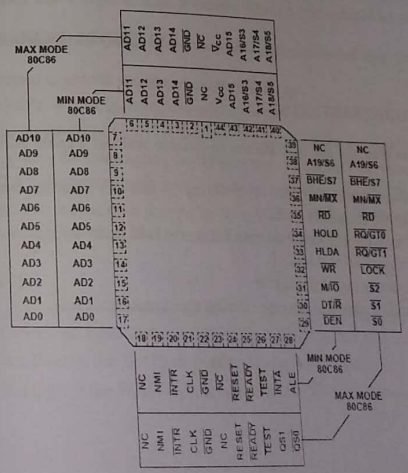
e.g. MOV AX, [BX+SI+05H]: 05H is 8-bit displacement

MOV AX, [BX+SI+1235H]:1235H is 16 bit displacement.

MOV [DI + BP + 20H], AX

PIN CONFIGURATION OF 8086 MICROPROCESSOR:





As shown in above pin configuration of 8086 microprocessor, the pins from 24-31 has a dual function, one for minimum mode and other for maximum mode. Their function depend on pin 33, i.e. MN / $\overline{MX}$, if MN / $\overline{MX}$ = 0, 8086 operates in maximum mode

MN / $\overline{MX}$ = 1, 8086 operates in minimum mode.

So, all the pins of 8086 microprocessor can be categorized into 3 sections as follows:

Section 1: The functions of pin remain same in both modes
Section 2: The functions of pin only in minimum mode
Section 3: The functions of pin only in maximum mode

Section 1(pins that remain same for both modes):

#AD7-AD0 [ADDRESS/DATA]

- The address/data bus line compose the multiplexed address data bus of the 8086 and contain the rightmost eight bits of the memory address or I/O port number whenever ALE is logic 1.
- It contains data whenever ALE is logic 0.
- These pins are at high impedance state during a hold acknowledgement.

#AD15-AD8 [ADDRESS/DATA]

- The 8086 address/data bus lines compose the upper multiplexed address/data bus on the 8086.
- These lines contain address bits A15-A8 whenever ALE is logic 1 and contains data when ALE is logic 0.
- These pins enter a high-impedance state whenever a hold acknowledgement occurs.

#A19/S6-A16/S3 [ADDRESS/STATUS]

- The address/status bus bits are multiplexed to provide address signals A19-A16 and also status bits S6-S3.
- These lines carry the address bits (A19-A16) during T1 of every machine cycle. T2 onwards these lines carry the status signals S6-S3.
- These pins also attain a high-impedance state during the hold acknowledge.
- Status bit S6 is always remains a logic 0
- Bit S5 indicates the condition of IF flag bit
- S4 and S3 show which segment is accessed during the current bus cycle.

S4	S3	FUNCTION
0	0	EXTRA SEGMENT
0	1	STACK SEGMENT
1	0	CODE SEGMENT
1	1	DATA SEGMENT

TABLE: FUNCTION OF STATUS BITS S3 AND S4

#$\overline{RD}$ [READ]

- Whenever the read signal is logic 0, the data bus is receptive to data from the memory or I/O devices connected to the system.
- This pin floats to its high-impedance state during a hold acknowledge.

#READY [READY]

- This input is controlled to insert wait states into the timing of the Microprocessor.
- If the READY pin is placed at logic 0 level, the MPU enters into wait states and remains idle.
- If the READY pin is placed at logic 1 level, it has no effect on the operation of the MPU.

#INTR [INTERRUPT REQUEST]

- Interrupt request is used to request a hardware interrupt.
- This is maskable, level-triggered, low-priority interrupt.
- On receiving an interrupt on INTR line, microprocessor executes two $\overline{INTA}$ pulses.
- First $\overline{INTA}$ pulse- the interrupting device prepares (calculates) the vector number.
- Second $\overline{INTA}$ – the interrupting device sends the vector number "N" to the microprocessor.
- Control shifts to the location pointed by (N x 4) in the IVT and CS and IP are loaded.
- If INTR is held high when IF=1, the 8086 enters an interrupt acknowledge cycle.

#$\overline{TEST}$ [TEST]

- The $\overline{TEST}$ pin is an input that is tested by the WAIT instruction.
- If $\overline{TEST}$ is a logic 0, the WAIT instruction functions as NOP.
- If $\overline{TEST}$ is a logic 1,the WAIT instruction waits for TEST to become a logic 0.

- This pin is most often connected to the 8087 Numeric Coprocessor.
- In Maximum mode, whenever the co-processor is busy, it makes this pin HIGH.
- In Minimum mode, it is connected to ground.

#NMI [NON MASKABLE INTERRUPT]

- The Non-Maskable, edge-triggered, high-priority Interrupt input is similar to INTR except that the NMI interrupt does not check to see whether the IF flag bit is logic 1.
- If NMI is activated, the microprocessor executes INT 2 and takes control to location 2 * 4 =00008H in the Interrupt Vector Table (IVT) to get the value of CS and IP.

#RESET [RESET]

- The RESET input causes the MPU to reset itself if this pin is held high for a minimum of four clocking periods.
- The 8284 Clock generator provides this signal.
- Whenever the 8086 is reset, it begins executing instructions at memory location FFFF0H and disables future interrupts by clearing the IF flag bit.

#CLK [CLOCK]

- The CLOCK pin provides the basic timing signals to the MPU.
- The clock signals must have a duty cycle of 33 percent (high for 1/3 clocking period and low for 2/3) to provide proper internal timing for the 8086.
- An external clock generator like 8284 provides the clock signal.

#VCC [POWER SUPPLY]

- The power supply input provides a +5.0V, ±10% signal to the MPU

#GND [GROUND]

- The GROUND connection is the return for the power supply.
- Note that the 8086 MPU have two pins labeled GND-both must be connected to ground for proper operation.

#MN/$\overline{MX}$ [MINIMUM/MAXIMUM]

- The MINIMUM/MAXIMUM mode pin selects either minimum mode or maximum mode operation for the MPU.
- If minimum mode is selected the MN/$\overline{MX}$ pin must be connected directly to +5.0 V.

#$\overline{BHE}$ /S7 [BUS HIGH ENABLE/STATUS]

- The Bus High Enable pin is used in 8086 to enable the most significant data bus bits (D15-D8) during a read or write operation.
- The state of S7 is always at logic 1.

Section 2(MINIMUM MODE PINS)

- Minimum mode operation of 8086 is obtained by connecting the MN/MX pin directly to +5V through a pull up resistor.

#M/$\overline{IO}$

- The M/$\overline{IO}$ (8086) pin select Memory or I/O.
- This pin indicates that the MPU address bus contains either a memory address of an I/O port address.
- This pin is at high impedance state during a hold acknowledge.

#$\overline{WR}$ [WRITE]

- The WRITE line is a strobe that indicates that the 8086 is outputting data to memory or I/O device.
- During the time that the WR is logic 0, the data bus contains valid data for memory or I/O.
- This pin floats to a high-impedance during a hold acknowledge.

#ALE [ADDRESS LATCH ENABLE]

- It shows that the 8086 address/data bus contains address information.
- This address can be a memory address or an I/O port number.
- The ALE signal does not float during a hold acknowledge.

#DT/$\overline{R}$ [DATA TRANSMIT / RECEIVE]

- It shows that the MPU data bus is transmitting or receiving.
- Data Transmitting if Logic 1.
- Data Receiving if Logic 0.
- This signal is used to enable external data bus buffers.

#DEN [DATA BUS ENABLE]

- It activates external data bus buffers.
- It is used to enable the data transceivers (bi-directional buffers- 8286)
- In Maximum mode, Bus Controller IC- 8288 gives the DEN signal.

#HOLD

- The HOLD input requests a DMA.
- If the HOLD signal is logic 1, the MPU stops executing s/w and places its address, data and control bus at the high-impedance state.
- If HOLD is set to 0, the MPU executes s/w normally.

#HLDA [HOLD ACKNOWLEDGE]

- Hold Acknowledge indicates that that 8086 has entered hold state.

#$\overline{SS0}$

- This signal is combined with IO/M and DT/R to decode the function of current bus cycle.

IO/$\overline{M}$	DT/$\overline{R}$	$\overline{SS0}$	DESCRIPTION
0	0	0	INTERRUPT ACK
0	0	1	MEMORY READ
0	1	0	MEMORY WRITE
0	1	1	HALT
1	0	0	OPCODE FETCH
1	0	1	I/O READ
1	1	0	I/O WRITE
1	1	1	PASSIVE

Section 3 (MAXIMUM MODE PINS)

- In order to achieve maximum mode for use with external coprocessors, connect MN/MX pin to ground.

#$\overline{S2}$, $\overline{S1}$ **AND** $\overline{S0}$

- The Status bits in 8086 indicate the function of the current bus cycle.
- These signals are normally decoded by the 8288 Bus Controller.

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	DESCRIPTION
0	0	0	INTERRUPT ACK
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

#$\overline{RQ}/\overline{GT1}$ **AND** $\overline{RQ}/\overline{GT0}$ **[REQUEST/GRANT]**

- The Request grant pins request DMA during maximum mode operation.
- These lines are bi-directional and are used to both request and grant a DMA operation.

#$\overline{LOCK}$

- The Lock output is used to lock peripherals of the system.

#QS1 AND QS0 [QUEUE STATUS BITS]

- The queue status bits show the status of the internal queue.

QS1	QS0	FUNCTION
0	0	Queue is idle
0	1	First byte of opcode
1	0	Queue is empty
1	1	subsequent byte of opcode

MINIMUM MODE VS MAXIMUM MODE

- There are two available modes of operation for the 8086: Minimum and Maximum Mode.
- Minimum mode operation is obtained by connecting the mode selection MN/MX to +5V.
- Maximum mode is selected by grounding MN/MX pin.

#MINIMUM MODE OPERATION

- The mode of operation provided by minimum mode is similar to that of 8085A , the most popular Intel 8 bit MPU.
- It is the least expensive way to operate the 8086
- It costs less because all the control signals for Memory and I/O are generated by MPU.
- It costs less because all the control signals are similar to those of Intel 8085A
- These control signals are similar to those of Intel 8085A
- 8288 Bus Controller is not required.

#MAXIMUM MODE OPERATION

- The Maximum mode is unique and designed to be used whenever a coprocessor exists in a system.(8087 Arithmetic Coprocessor)
- It requires additional external bus controller : 8288
- This 8288 Bus Controller is required because there are not enough pins on the 8086 for bus control during Maximum mode.

INSTRUCTION SETS 8086:

1. DATA TRANSFER INSTRUCTIONS

1.1 GENERAL PURPOSE BYTE OR WORD TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS

MOV
 	
MOV Destination, Source
MOV CX,0037H ; CX ← 0037H
MOV BL,[4000H] ; BL ← DS:[4000H]
MOV AX,BX ; AX← BX
MOV DL, [BX] ; DL ← DS : [BX]
MOV DS,BX ; DS ← BX
	Copy byte or word from specified source to specified destination.
	
	Source: Register, Memory Location, Immediate Number,
	Destination: Register, Memory Location.
	
	Both, source and destinations cannot be memory locations

PUSH **PUSH Source** PUSH BX ; SS:[SP-1] ← BH, SS:[SP-2] ← BL, SP ← SP-2	Copy specified source (word) to top of stack and decrement the stack pointer by two. - The source must be a word (16 bits). **Source** : Register, Memory Locations
POP **POP Destination** POP CX ;CL ← SS:[SP], CH← SS:[SP+1] POP DS	Copy word from top to stack to specified location and increment the stack pointer by 2. **Destination:** Register (except CS),Memory Location

Note: MOV, PUSH & POP are only instructions that use segment registers (except CS) as Operands.	
XCHG **XCHG Destination, Source** XCHG AX,BX ; AX ⟷ CX XCHG BL,CH ; BL ⟷ CH	Exchange word or byte between the source and destination specified in the instruction. **Source**: Register, Memory Location **Destination**: Register, Memory Location. Both, source and destinations cannot be memory locations
XLAT E.g. XLAT ; AL ← DS :[BX+AL]	Translate a byte in AL using a table in memory. It first adds AL + BX to form memory address. It then copies the content into AL

1.2 SIMPLE INPUT AND OUTPUT PORT TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS
IN	Copy a byte or word from specified port to accumulator.

IN AX, 16-bit Port_Addr (direct) or DX register (Indirect), or **IN AH, 8-bit Port_Addr,** or **IN AL, 8-bit Port_Addr** IN AL,34H ; AL ← [34H]i/o, where I/O port 34H is 8-bit port IN AX,80H IN AL,DX IN AX,DX	
OUT **OUT Port_Addr, AX** OUT 2CH,AX ; OUT DX,AX ; [DX]i/o ← [AX] as I/O port pointed by DX is 16-bit port	Copy a byte or word from accumulator to specified port.

1.3 SPECIAL ADDRESS TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS

LEA **LEA Register, Source** E.g. LEA BX,PRICE ; BX ← Effective address of PRICE in Data Segment	Load effective address (offset address) of operand into specified register.
LDS **LDS Register, Source** E.g. LDS BX,[4326H] ;BX ← {DS:[4326], DS:[4326+1] } , DS ← {DS:[4326 +2], DS:[4326+3] }	Load DS register and other specified register from memory.
LES	Load ES register and other specified register from memory.

1.4 FLAG TRANSFER INSTRUCTIONS

INSTRUCTIONS	COMMENTS
LAHF	Copy to AH with the low byte of the flag register.
SAHF	Stores AH register to low byte of flag register.
PUSHF	Copy flag register to top of stack and decrement the stack pointer by 2.

POPF	Copy word at top of stack to flag register and increment the stack pointer by 2.

2. ARITHMETIC INSTRUCTIONS

2.1 ADDITION INSTRUCTIONS

INSTRUCTIONS	COMMENTS

ADD
 	
ADD Destination, Source
 	
ADD AL,74H ; AL ← AL +74H
ADD BX,CX
	Add specified byte or word in source to the byte or word in destination and stores the result in destination.
	
	Both source & destination must be of same size.

ADC **ADC Destination, Source** ADC CL,BL	Add byte + byte + carry flag Add word+word + carry flag
INC **INC Register** INC CX INC BL	Increment specified byte or word by 1. Carry Flag is not affected.

AAA E.g $(3)_{10} = (33)_{ASCII}$ – [AL] $(5)_{10} = (35)_{ASCII}$ – [CL] ADD AL,CL ; Result =[AL]= 68 AAA ; After AAA instruction Result = [AH]=00 ; [AL] = 08	- ASCII adjust after addition. - It is used to get result in unpacked BCD form after adding 2 unpacked decimal numbers in ASCII form. - This instruction works only on AL.

2.2 SUBTRACTION INSTRUCTIONS

INSTRUCTIONS	COMMENTS
SUB **SUB Destination, Source** SUB CX,BX	Subtract byte from byte or word from word.
SBB	Subtract byte and carry flag from byte. Subtract word and carry flag from word.

SBB Destination, Source SBB CH,AL	
DEC **DEC Register** DEC CX	Decrement specified byte or word by 1.
NEG **NEG Register** NEG AL	Form 2's complement.
CMP **CMP Destination, Source** 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 E.g $(9)_{10} = (39)_{ASCII}$ – [AL] $(5)_{10} = (35)_{ASCII}$ – [CL] SUB AL,CL ; Result =[AL]= 04	- ASCII adjust after subtraction. - Used to get result in Unpacked BCD form after subtraction of 2 unpacked decimal number in ASCII form. - This instruction works only in AL. -

AAS ; After AAS instruction Result = C_y=0 ; [AL] = 04	
DAS E.g. [AL=](86)$_{10}$ + [CL=](57)$_{10}$ = (29)$_{10}$ SUB AL,CL ; Result – C_y= 0, [AL]=2F DAS ; After DAS instruction C_y= 0; [AL] = 29 So result = +(29)$_{10}$	- Decimal adjust after subtraction. - Get result in BCD form after performing BCD subtraction. - Works only on AL. -

AAM – BCD
AAS – ASCII
AAA – ASCII

2.3 MULTIPLICATION INSTRUCTIONS

INSTRUCTIONS	COMMENTS
MUL **MUL Source (unsigned 8/16 bit register)** MUL BL ; AX ← AL x BL MUL BX ; DX-AX ← AX x BX	Multiply unsigned byte by byte or unsigned word by word. If the source is 8-bit, it is multiplied with **AL** and the result is stored in **AX** (AH-higher byte, AL-lower byte). If the source is 16-bit, it is multiplied with **AX** and the result is stored in **DX-AX** (DX-higher byte, AX-lower byte). Source : Register, Memory Location MUL affects AF, SF, PF, ZF
IMUL **IMUL Source** IMUL CX	Multiply signed byte by byte or signed word by word.
AAM E.g. [AL=](05)$_{unpacked\ BCD}$ X [CL=](09)$_{unpacked\ BCD}$ = (0405)$_{unpacked\ BCD}$	- It is not really ASCII adjust after multiplication but it is BCD adjust after multiplication - It is used to get result in unpacked BCD form after performing multiplication of 2 unpacked BCD numbers. - This instruction works only on AX.

MUL CL ; Result – [AH]= 00, [AL]=2D AAM ; After AAM instruction [AH]=04; [AL] = 05 So, result = $(0405)_{Unpacked}$ $(45)_{packed}$	

2.4 DIVISION INSTRUCTIONS

INSTRUCTIONS	COMMENTS
DIV ***DIV Source (unsigned 8/16-bit register - Divisor)***	Used for Unsigned division. Divides a **WORD** by a **BYTE** or **DOUBLE WORD** by a **WORD.** If the **divisor is 8-bit** then the **dividend is in AX** register. - After division, the **quotient is in AL** and **remainder in AH** If the **divisor is 16-bit** then the **dividend is in DX-AX** register. - After division, the **quotient is in AX** and **remainder in DX**

DIV BL ; AX ÷ BL, AL ← quotient, AH ← remainder DIV BX ;{DX,AX} ÷ BX, AX ← quotient, DX ← remainder	Source: Register, Memory Location All flags are **Undefined** after DIV instruction. **Note: If the divisor is 0, or the result is too large to fit in AL(or AX in 16 bit divisor) then 8086 does a Type 0 Interrupt (Zero Divide Error)**
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 NOT BX	Invert each bit of a byte or word.
AND	AND each bit in a byte/word with the corresponding bit in another byte or word.

AND Destination,Source AND BH,CL	
OR OR Destination,Source OR AH,CL	OR each bit in a byte or word with the corresponding bit in another byte or word.
XOR XOR Destination,Source XOR CL,BH	XOR each bit in a byte or word with the corresponding bit in another byte or word.
TEST TEST Destination,Source TEST AL,BH	AND operands to update flags, but don't change the operands.

3.2 SHIFT INSTRUCTIONS

INSTRUCTIONS	COMMENTS
SHL/SAL SAL Destination,Count SHL Destination,Count e.g. SAL AL,01; Rotates content of AL, each bit 1 times left as: 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. - Here, sign bit is preserved.

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 RCL Destination,Count	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
JA/JNBE	Jump if Above/Jump if Not Below or Equal.
JAE/JNB	Jump if Above or Equal/Jump if Not Below.
JB/JNAE	Jump if Below/Jump if Not Above or Equal.
JBE/JNA	Jump if Below or Equal/Jump if Not Above.
JC	Jump if Carry Flag CF=1.
JE/JZ	Jump if Equal/Jump if Zero Flag (ZF=1).
JG/JNLE	Jump if Greater/Jump if Not Less than or Equal.

Microprocessors- Chapter-4
Prepared By: Er. Ramu Pandey, Asst. Prof. , NCIT

JGE/JNL	Jump if Greater than or Equal/Jump if Not Less than.
JL/JNGE	Jump if Less than/Jump if Not Greater than or Equal.
JLE/JNG	Jump if Less than or Equal/Jump if Not Greater than.
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)

4.3 ITERATION CONTROL INSTRUCTIONS

INSTRUCTIONS	COMMENTS
LOOP	Loop Through a Sequence of Instructions Until CX=0.

LOOPE/LOOPZ	Loop Through a Sequence of Instructions While ZF=1 and CX!=0.
LOOPNE/LOOPNZ	Loop Through a Sequence of Instruction While ZF=0 & CX!=0.
JCXZ	Jump to Specified Address if CX=0.

4.4 INTERRUPT INSTRUCTIONS

INSTRUCTIONS	COMMENTS
INT	Interrupt Program Execution if OF=1
INTO	
IRET	Return From Interrupt Service Procedure to Main Program.

5 PROCESSOR CONTROL INSTRUCTIONS

5.1 FLAG SET/CLEAR INSTRUCTION

NSTRUCTIONS	COMMENTS
STC	Set Carry Flag CF to 1.
CLC	Clear Carry Flag to 0.
CMC	Complement the State of CF.

STD	Set Direction Flag to 1.
CLD	Clear Direction Flag to 0.
STI	Set Interrupt Flag to 1. (Enable INTR Input).
CLI	Clear Interrupt Enable to 0

5.2 NO OPERATION INSTRUCTION

INSTRUCTIONS	COMMENTS
NOP	No Action Except Fetch and Decode.

5.3 EXTERNAL HARDWARE SYNCHRONIZATION INST.

INSTRUCTIONS	COMMENTS
HLT	Halt (Do Nothing) Until Interrupt or Reset.
WAIT	Wait Until Signal On the TEST Pin is Low.
ESC	Escape to External Coprocessor Such as 8087 or 8089.
LOCK	Prevents Another Processor From Taking the Bus

	While the Adjacent Instruction Executes.

6 STRING INSTRUCTIONS

INSTRUCTIONS	COMMENTS
REP	Repeat Instruction Until CX=0.
REPE/REPZ	Repeat if Equal/Repeat if Zero
REPNE/REPNZ	Repeat if Not Equal/Repeat if Not Zero.
MOVS/MOVSB/MOVSW E.g. MOVSB ; Moves content of memory location pointed by DS: [SI] to ES:[DI] After executing string instruction if direction flag is reset then content of DI is incremented by 1 and vice versa.	Move Byte or Word From One String to Another.
COMPS/COMPSB/COMPSW	Compare Two String Bytes or Two String Words.

SCAS/SCASB/SCASW	- Compares a Byte in AL or Word in AX With a Byte or Word Pointed By DI in ES. - Subtract content of memory locations ES:[DI] from AL. - Data will not change , only flags are affected.

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]

Microprocessors- Chapter-4
Prepared By: Er. Ramu Pandey, Asst. Prof. , NCIT

[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

END [PROCEDURE NAME]

Microprocessors- Chapter-4
Prepared By: Er. Ramu Pandey, Asst. Prof. , NCIT

#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

- **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

Microprocessors- Chapter-4
Prepared By: Er. Ramu Pandey, Asst. Prof. , NCIT

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
```

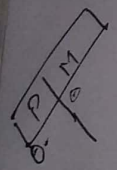
Now the macro is called in the main program i.e. macro invocations is as shown below:

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

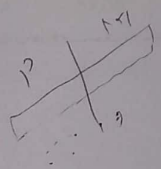
Macro Vs Procedure:

(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

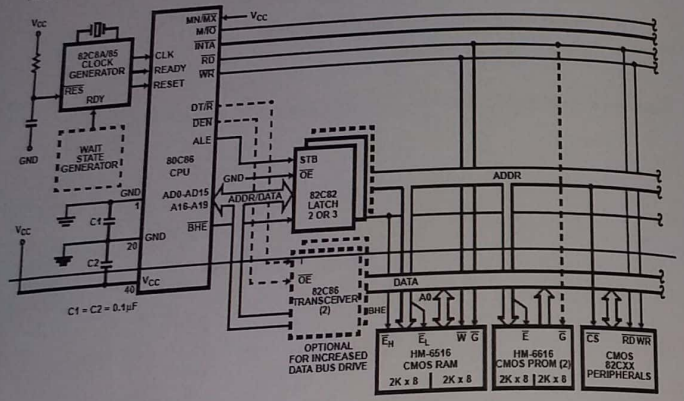
8086 System Timing Diagram

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:

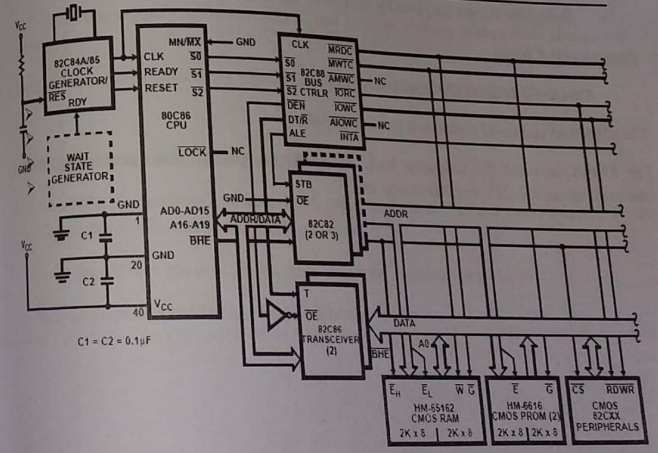
- $\overline{DEN}$ of 8086 is given to $\overline{STB}$ of 8286,
- DT/$\overline{R}$ of 8086 is given to T of 8286,
- When $\overline{DEN}$ = 0, DT/$\overline{R}$ =1, then 8086 is transmitting data.

- 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



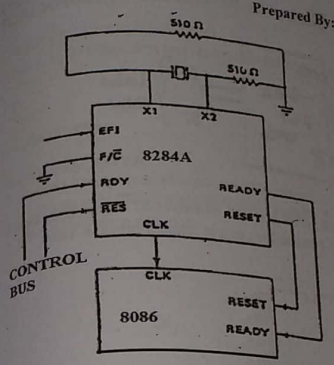
- 8086 Works in Maximum mode when MN/$\overline{MX}$ = 0
- In Maximum mode, there is at least one more processor in the system besides 8086
- Clock is provided by 8284 Clock Generator
- The most significant part of the Maximum mode mode circuit is **8288 Bus Controller**
- Instead of 8086, the bus controller 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.



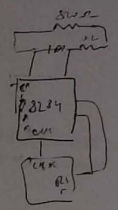


Fig. Connection of 8284A with 8086 microprocessor

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.

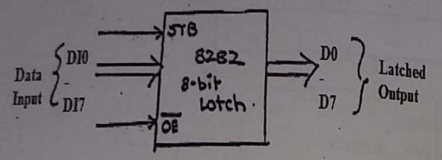
. By connecting crystal oscillator across X_1 and X_2. Now we have to keep F/ $\overline{C}$ (Frequency / Crystal) = 0.

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.

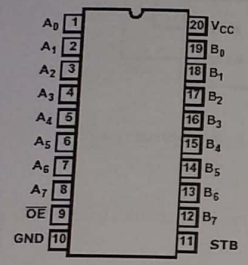


Figure: 8282 Address Latch

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.
-

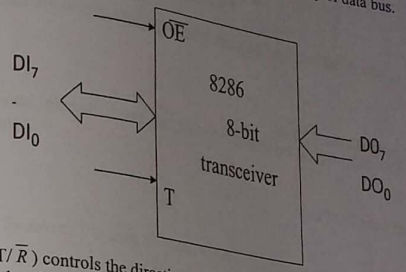
 # $\overline{OE}$

- 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.

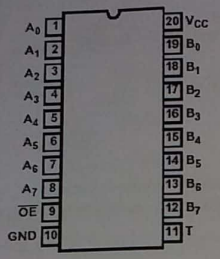


Figure: 8286 Transceiver

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.

Microprocessors- Chapter-4
Prepared By: Er. Ramu Pandey, Asst. Prof. , NCIT

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

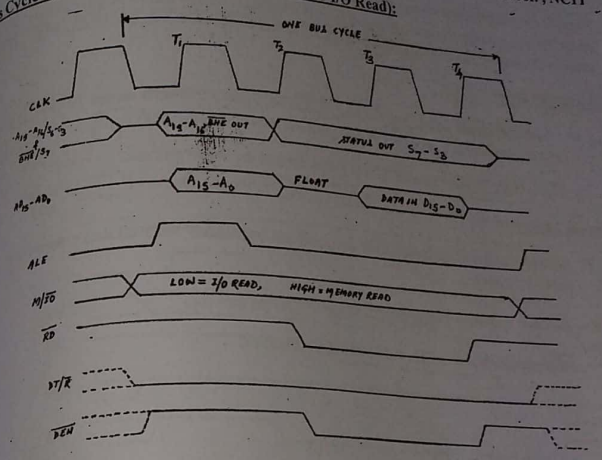


Fig. Bus Cycle For Minimum mode (I/O Read or Memory Read)

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.

Microprocessors- Chapter-4
Prepared By: Er. Ramu Pandey, Asst. Prof. , NCIT

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

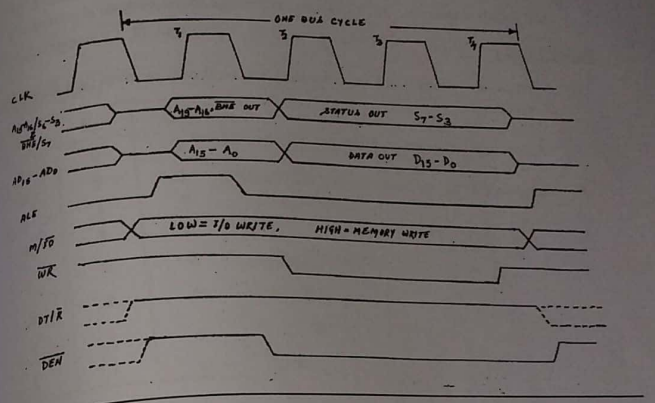


Fig. Bus Cycle For O/P Operation (Memory Write or I/O Write)

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.

Microprocessors- Chapter-4
Prepared By: Er. Ramu Pandey, Asst. Prof. , NCIT

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:

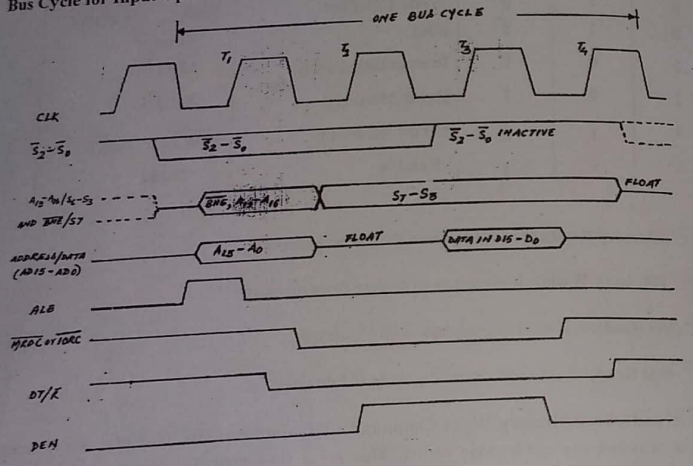


Fig. 8086 Maximum mode for Input Operation (I/O Read or Memory Read)

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:

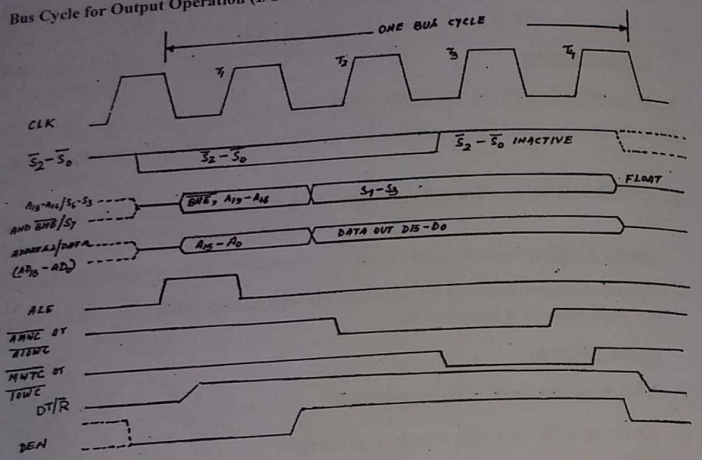


Fig. 8086 Maximum mode for Input Operation (I/O Write or Memory Write)

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.

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
```

```
main endp ; ends the procedure with name 'main'
end main     ; ends the program
```

2.

```
title program to Print the Sum of Two 8 Bit Numbers
dosseg
.model small
.stack 100h
.data
    val1 db 89
    val2 db 10
    msg db 'sum of 2 numbers : $'
.code
    main proc
    mov ax,@data
    mov ds,ax
    mov ax,0
    mov al,val1
    add al,val2
    aam          ;aam Converts Binary Value to Unpacked BCD.
    add ax,3030h  ;ax is Added with 3030H to Obtain ASCII Value
    push ax
;;;display message
```



```
lea dx,msg
mov ah,09h
int 21h
;;;end display message
pop ax
mov dl,ah
mov dh,al
mov ah,02h
int 21h
mov dl,dh
mov ah,02h
int 21h
mov ax,4c00h
int 21h
main endp
end main
```

3.

```
title Program to add numbers input from keyboard
.model small
.stack
.data
.code
main proc
mov ax, @data
mov ds,ax

mov ah,01h
```

```
int 21h
sub al,30h
mov bl,al      ;store 1st number in bl register
mov ah,01
int 21h
sub al,30h
add al,bl
push al
jnc SKIP
mov dl, '1'
mov ah,02h
int 21h
    SKIP: pop al
    add al,30h
mov dl,al
mov ah,02h
int 21h
mov ah,4ch
int 21h
main endp
end main
```


Multiplication:

Multiplicand	Multiplier	Product
AL	8 bit	AX
AX	16 bit	DX:AX

Division:

Dividend	Divisor	Quotient	Remainder
AX(16 bit)	8 bit	AL	AH
DX:AX(32 bit)	16 bit	AX	DX

title 8-bit multiplication

```
dosseg
.model small
.stack 100h
.code
   main proc
   mov ax,@data
   mov ds,ax
   mov al,05h  ; multiplier must be in 'al'
   mov bl,10h ;multiplicand can be in any register
   mul bl        ;result in ax
   aam           ; aam is 8086 instruction that converts binary value to unpacked bcd
   add ax,3030h  ; ax is added with 3030h to obtain ascii value
   mov ax,4C00h
```

```
    int 21h
    main endp
end main
```

5.

```
title 16-bit by 8-bit division
dosseg
.model small
.stack 100h
.data
val1 dw 0083h  ; dw (data word) for 16-bit data
val2 db 02h    ;db (data byte) for 8-bit data
.code
  main proc
  mov ax, @data
  mov ds,ax
  mov ax,val1 ;dividend must be in 'ax' register
  div val2    ; quotient in 'al' and remainder in 'ah' register
  mov ax,4C00h
  int 21h
  main endp
end main
```



```
title program to transfer a block of data from one part of the memory to another
dosseg
.model small
.data
    array1 db 22h, 33h, 44h, 55h, 66h
    array2 db 5 dup(?)
.code
main proc
    mov ax,@data
    mov ds,ax
    mov si,offset array1 ;starting offset address of array1 passed to 'si'.
    mov di,offset array2
    mov cx,0005h    ; counter
again:
    mov al,[si]     ;[ds:si] content copied to al
    mov [di],al     ;[al] content copied [ds:di]
    inc ai
    inc di
    loop again      ;loop continues till cx=0
    mov ah,4ch
    int 21h
main endp
end main
```

7.

title program to find the largest number.

```
dosseg
.model small
.stack 100h
.data
        array db 61h,43h,45h,42h,41h
        great db 00h
.code
main proc
        mov ax,@data
        mov ds,ax
        mov cx,0005h ;counter set to 5 because array has 5 data
        mov si,offset array
        mov bl,great
again:
        cmp bl,[si]     ;compare bl and [si]
        jnc nochange    ;if no carry then content of bl is greater.
        mov bl,[si]     ;if carry occurs then content of bl is
                        smaller than that of content of [si]
nochange:
        inc si
        loop again
        mov dl,great
        mov ah,02h
        int 21h
        mov ah,4ch
        int 21h
main endp
end main
```


title program to find the smallest number

```
dosseg
.model small
.stack 100h
.data
        array db 61h,43h,45h,42h,41h
.code
main proc
        mov ax,@data
        mov ds,ax
        mov cx,0005h
        mov si,offset array
        mov bl,small
again:
        cmp bl,[si]
        jc nochange
```

```
        limit db 40
        timeschk db 38
        multiplier db 2
        result db 30 dup(?)
.code
begin proc
        mov ax,@data
        mov ds,ax
        mov ax,0
        lea si,array            ;need to  multiply with 2,3,5,7
        lea di,limit            ;decreased by 1 each time checking a number is
completed
        lea bp,result           ;to store the non prime numbers
        mov cl,timeschk         ;no. of numbers to check.i.e.above
2(smallest prime)
        mov bl,multiplier       ;increase each time to compare result
        mov dl,timesmul         ;times to multiply 2,3,5 and 7

big:
        jmp prime               ;check prime or not
                                ;check complete
small:
        mov bl, multiplier      ;bl = 2 again
        mov dl,timesmul         ;dl=times to mul
        sub [di],1              ;decreasing number to check
loop big
        jmp finish
```



```
prime:
        mov al,[si]             ;mov 2 in al
        mul bl                  ;bl has 2 ; al becomes 4
        cmp al,[di]             ;compare al with number to check prime
        je notprime
        add bl,1                ;inc no. to multiply
        sub dl,1                ;dec times to multiply; like count
        jnz prime               ;loop till dl=0
        inc si                  ;get 3 into si,next no to multiply
        mov dl,timesmul         ;dl=times to mul
        mov bl,multiplier       ;bl=2 again
        inc si                  ;si point to no.5
        mov al,[si]             ;check if array content is over; si pointing to di ?
        dec si                  ;si points to 3 ,next number to multiply
        cmp al,[di]             ;is al=[di] ?
        jne prime
        mov [bp],al
        inc bp
        lea si,array
        jmp small

notprime:
        lea si,array            ;make si point to array
```

```
        jmp small
finish: mov cx,30
        lea bp,result
        getnonprime:
            mov al,[bp]
            inc bp
            loop getnonprime
        mov ax,4c00h
        int 21h
begin endp
end begin
```

11.

title 'hello world 'string display program

```
 dosseg
.model small
.stack 100h

.data
    HelloMessage DB 'Hello World',13,10,'$'
.code
main proc
mov  ax,@data
mov  ds,ax          ;set DS to point to the data segment
mov  ah,09          ;DOS print string function
mov  dx,OFFSET HelloMessage ;point to "Hello, world"
int  21h            ;display "Hello, world"
mov  ah,4ch         ;DOS terminate program function
int  21h            ;terminate the program
main endp
end main
```


title 'hello world ' string display program without using function 09h

```
dosseg
.model small
.stack 100h
.code
main proc
    mov ax,@data
    mov ds,ax
    mov cx,len
    mov si,offset string

again:
    mov dl,[si]
    mov ah,02h
    int 21h
```

mov ax,@data
mov ds,ax
mov si,offset s2
Again:
mov dl,[si]
mov ah,02h
int 21

```
        inc si
        loop again

        mov ah,4ch
        int 21h
main endp
.data
        string db 'This is a string','$'
        len dw $-string-1
end main
```

13.

title program to print a defined string in reverse order

```
dosseg
.model small
.stack 100h
.data
        string db 'This is a string.','$'
        len dw $-string-1
.code
main proc
        mov ax,@data
        mov ds,ax
        mov cx,len
```



```
        mov si,offset string
        add si,len
        dec si
again:
        mov dl,[si]     ;the character to be outputted must be in dl
        mov ah,02h      ;character output function
        int 21h
        dec si
        loop again
        mov ah,4ch
        int 21h
main endp
endp main
```

14.

title Program to display the reverse of the string entered from the keyboard.

```
.model small
.stack 64h
.data
string1 db 'Enter the string  :$'

indata  label byte
maxlen  db   20
actlen  db   ?
```

```
kbinput db  20 dup ('$')

string2 db 'Reverse string is :$'
result db 20 dup ('$')
.code
begin proc far
      mov ax,@data
      mov ds,ax
      mov es,ax
      call clearscreen  ;clear screen and set the attribute
      mov dx, 0201h
      call setcursor    ; set the cursor position
      call display1     ; display the output
      call result1
      call display2
      mov ax,4c00h
      int 21h
begin endp

clearscreen proc near
      mov ax,0600h      ; request scroll screen
      mov bh,04h        ;attribute
      mov cx,0000       ;from 00,00
      mov dx,184fh      ;to 24,79
```



```
      int 10h
      ret
      clearscreen endp

setcursor proc near
      mov ah,02h
      mov bh,00
      int 10h
      ret
      setcursor endp

display1  proc near
      push dx
      lea dx,string1
      mov ah,09h
      int 21h

      mov ah,0Ah
      lea dx,indata
      int 21h
      pop dx
      ret
      display1 endp
```

```
result1 proc near
        lea di,result
        lea si,indata
        inc si
        mov cl,[si]
        mov ch,00
        add si,cx
back:   mov bl,[si]
        mov [di],bl
        inc di
        dec si
        dec cl
        jnz back
        mov bl,'$'
        mov [di],bl
        ret
        result1 endp


display2 proc near
        inc dh
        call setcursor
        push dx
        mov ah,09h
```

Microprocessors- Chapter-4
Prepared By: Er. Ramu Pandey, Asst. Prof. , NCIT

```
        lea dx, string2
        int 21h
        pop dx
        mov ah,09h
        lea dx,result
        int 21h
        ret
        display2 endp


end begin
```

title program to print each word of a string in different lines

```
dosseg
.model small
.stack 100h
.code
main proc
        mov ax,@data
        mov ds,ax
        mov si,offset string
again:
```

```
        mov dl,[si]     ;the character to be outputted must be in dl
        cmp dl,24h      ;ASCII value of $ is 24h
        jz last         ;when gets $ sign, program jumps to
                         label- last for termination
        cmp dl,20h      ;ASCII value of space is 20h
        jz nextline     ; when gets space in string jumps to
                         label- nextline
        mov ah,02h      ; character output function
        int 21h
        inc si
        jmp again
nextline:
        inc si
        mov dl,0dh      ;cariage return or effect of enter key
                         pressed
        mov ah,02h
        int 21h
        mov dl,0ah      ;start in next line
        mov ah,02h
        int 21h
        jmp again
last:
        mov ah,4ch
        int 21h
main endp
```



```
.data
        string db 'I like Programming','$'
end main
```

title ALP that asks user to 'Input password' and compares the entered password and displays 'Welcome' if it is equal to 'ncit' otherwise display 'Invalid User'.

```
.model small
.stack
.data
  password db 'ncit','$'
  msg1 db 'Insert Password ','$'
  msg2 db 10,13,'Welcome','$'
  msg3 db 10,13,'Invalid User','$'
  val  db  ?
.code
  main proc
     mov ax,@data
     mov ds,ax

     mov dx,offset msg1
     mov ah,09h
     int 21h
```

```
    mov bh,0
    mov cx,4
    mov si,offset password
AGAIN: mov ah,01h
    int 21h
    mov val,al
    ;mov dl,'*'
    ;mov ah,02h
    ;int 21h
    mov bl,[si]
    cmp val,bl
    jne SKIP
    inc bh
SKIP:inc si
    loop AGAIN
    cmp bh,4
    jne INVALID

    mov dx,offset msg2
    mov ah,09h
    int 21h
    jmp EXIT
INVALID: mov dx,offset msg3
    mov ah,09h
```



```
    int 21h
EXIT:mov ah,4ch
    int 21h
    main endp
    end main
```

title alp to display the vowels present in the input string

```
.model small
.stack 100h
.data
inpstr db 255 dup(?)
oustr db 13,10;define string
.code
main proc
        mov ax,@data
        mov ds,ax
        lea si,inpstr
        lea di,oustr
        mov cx,0
        check: mov ah,1
        int 21h
        cmp al, 0dh
        je exit
        cmp al,'a' ; compare with 'a'
```

```
je a1; if equals goto a1
jne e1; else goto e1 to check for next vowel
jmp check
a1:
 mov [di],al; push the contentd to di if char is equal to 'a'
inc cx
inc si
inc di
jmp check ; goto check to take anoother character
e1:
 cmp al,'e'; compare with 'e'
je ee
jne i1
ee:
mov [di],al ;push the content to di if char is equal to 'e'
inc cx
inc si
inc di
jmp check
i1:
cmp al,'i'
je ii
jne o1
ii:
```



```
mov [di],al ;push the content to di if char is equal to 'i'
inc cx
inc si
inc di
jmp check
o1:
 cmp al,'o'
 je oo
 jne u1
 oo:
  mov [di],al ;push the content to di if char is equal to 'o'
  inc cx
  inc si
  inc di
  jmp check
  u1:
   cmp al,'u'
   je oo
   jne check
   uu:
    mov [di],al; push the content to di if the char is equal to 'u'
    inc cx
    inc si
    inc di
```

```
    jmp check
    exit:
    call showvowels; call the procedure display it
    mov ax,4c00h
    int 21h
    main endp
    showvowels proc
    mov ah,02
    mov dh,10
    mov dl,30
    mov bh,00 ; load page nunmber
    int 10h
    lea si ,oustr
    add si,cx
    mov [si],"$" ; make $ a string
    sub di,cx
    mov ah,9h
    mov bl,11001011b
    int 10h
    mov dx,offset oustr
    int 21h
    ret
    showvowels endp
end main
```


title to input string and count number of vowel

```
dosseg
.model small
.stack 100h
.data
    string db 99 dup(?)
    count db 00h

.code
main proc
mov ax,@data
mov ds,ax
mov si, offset string
mov di,offset count
 again:
     mov ah,01h
     int 21h
     cmp al,0dh
     jz down
      mov [si],al
```

```
        inc si
        inc bl
        jmp again
down:

        mov dl,0dh
        mov ah,02h
        int 21h
        mov dl,0ah
        mov ah,02h
        int 21h
        mov cl,bl
        mov ch,00h
        mov si,offset string
again2:
        mov dl,[si]
        cmp dl,20h
        jz next
        mov al,[si]
        cmp al,'a'
        jz count1
        cmp al,'A'
        jz count1
        cmp al,'e'
```



```
        jz count1
        cmp al,'E'
        cmp al,'i'
        jz count1
        cmp al,'I'
        jz count1
        cmp al,'o'
        jz count1
        cmp al,'O'
        jz count1
        cmp al,'u'
        jz count1
        cmp al,'U'
        jz count1
        jmp jump

count1:
        inc bh

jump:
        mov dl,al
        mov ah,02h
        int 21h
        inc si
```

```
        loop again2
        jmp last


next:  inc count
        inc si
        mov dl,0dh
        mov ah,02h
        int 21h
        mov dl,0ah
        mov ah,02h
        int 21h
        dec cx
        jmp again2
last:
        inc si
        mov [si],bh
        mov dl,count
        mov ah,02h
        mov ax,4c00h
        int 21h


main endp
end main
```

Microprocessors- Chapter-4
Prepared By: Er. Ramu Pandey, Asst. Prof. , NCIT

```
title program to change case of the letter on given string
.model small
.stack 100h
.data
str1 db "How are You"
str2 db 11 dup (?)
.code
main proc
        mov ax,@data
        mov ds,ax
        lea si,str1
        lea di,str2
        mov cx,11d ; count and store the length of given string in cx
case:
        cmp cx,0000h
        je exit
        mov ah,[si]
        cmp ah,60h ;ascii value of small letters starts from 61h i.e. ascii value of 'a'=61h
        ja small  ; if the character has ascii value above 60h then it is small letter
        cmp ah,5Ah ; ascii values of capital 'Z' is 5Ah
        jb cap ; if the character has ascii value below 5Ah then it is capital letter
small:
```

```
        and ah,11011111b
        mov [di],ah
        inc si
        inc di
        dec cx
        jmp case
cap:
        or ah, 00100000b
        mov [di],ah
        inc si
        inc di
        dec cx
        jmp case
exit:
        call display
        mov ax,4c00h
        int 21h
main endp
display proc
        lea si,str2
        add si,12d
        mov [si],"$"
        sub di,12d
        mov ah,9h
```

Microprocessors- Chapter-4
Prepared By: Er. Ramu Pandey, Asst. Prof. , NCIT

```
        lea dx,str2
        int 21h
        ret
display endp
end main
```

title to input string and display in different line and convert lowercase to uppercase

```
dosseg
.model small
.stack 100h
.data
    string db 99 dup(?)
    count db 00h

.code
main proc
mov ax,@data
mov ds,ax
mov si, offset string
mov di,offset count

again:
```

```
        mov ah,01h
        int 21h
        cmp al,0dh
        jz down
        mov [si],al
        inc si
        inc bl
        jmp again
down:

        mov dl,0dh
        mov ah,02h
        int 21h
        mov dl,0ah
        mov ah,02h
        int 21h
        mov cl,bl
        mov ch,00h
        mov si,offset string
again2:
        mov dl,[si]
        cmp dl,20h
        jz next
        mov al,[si]
        cmp al,'a'
        jc jump
        cmp al,'z'+1
        jnc jump
        sub al,20h
jump:
        mov dl,al
        mov ah,02h
        int 21h
        inc si
        loop again2
        jmp last

next:   inc count
        inc si
        mov dl,0dh
        mov ah,02h
        int 21h
        mov dl,0ah
        mov ah,02h
        int 21h
        dec cx
        jmp again2
```

```
last:
    mov ax,4c00h
    int 21h

    main endp
    end main
```

21.

title program to input a string and print each word in different lines

```
dosseg
.model small
.stack 100h
.code
main proc
    mov ax,@data
    mov ds,ax
    mov si,offset string
    again:
    mov ah,01h
    int 21h
    cmp al,0dh
    jz down
    mov [si],al
    inc si
    inc bl
```

Microprocessors- Chapter-4
Prepared By: Er. Ramu Pandey, Asst. Prof. , NCIT

```
    jmp again
    down:
    mov dl,0dh
    mov ah,02h
    int 21h

    mov cl,bl
    mov ch,00h
    mov si,offset string
    again2:
    mov dl,[si]
    cmp dl,20h
    jz nextline
    mov ah,02h
    int 21h
    inc si
    loop again2
    jmp last
    nextline:
    inc si
    mov dl,0dh
    mov ah,02h
    int 21h
    mov dl,0ah
```

```
 mov ah,02h
 int 21h
 mov dl,0ah
 int 21h
 dec cx
 jmp again2
 last:
 mov ah,4ch
 int 21h
main endp
.data
      string db 99 dup(?)
end main
```