UNIT-1 : OVERVIEW OF 8086 μP

1. Introduction to 8086
2. Architecture of 8086
3. Memory segmentation
4. Registers of 8086
5. Pin configuration of 8086
6. Min. mode operation with Timing diagram
7. Max. mode operation with Timing diagram
8. Physical memory organization and Memory banks accessing
9. Interrupts of 8086 and Interrupt Vector Table (IVT)

1.1. INTRODUCTION TO 8086

The Intel released 8086 microprocessor in 1978, which is fabricated using HMOS technology. The features of Intel 8086 are given below

- The 8086 is a 16-bit microprocessor
- It has 16-bit ALU
- It has 16-bit data bus and 20-bit address bus
- It can address 2^{20} locations i.e., 1 M Bytes of memory
- It uses memory segmentation
- It uses 2-stage pipeline concept – Fetch and Execute
- It has 6-bytes of Instruction Queue
- It is packaged in 40-pin DIP
- It is available in three clock rates : 5 MHz, 10 MHz and 8 MHz
- It can be operated in two modes: *Min. mode and Max. Mode*
- The 8086 is operated in Minimum mode when only one processor is to be used in a microcomputer system. The Maximum mode is used when more than one processor are used in a system.

Note: The architecture of 8088 is similar to 8086 except for two changes:

- The 8088 has 4-byte instruction queue where as 8086 has 6-byte instruction queue.
- The 8088 has 8-bit data bus where as 8086 has 16-bit data bus
- There is no $\overline{BHE}$ signal in 8088

1.2. ARCHITECTURE OF 8086

The complete architecture of 8086 is divided into two parts

(A) Bus Interface Unit (BIU) and
(B) Execution Unit (EU).

The Bus Interface Unit is responsible for all read and write operations of Memory and I/O. It sends out 20-bit Physical address, fetches instruction code bytes from memory and stores them in Instruction Queue.

The Execution unit is responsible for decoding and executing the instructions. It receives instruction code bytes from Instruction Queue in BIU, decodes and executes them.

These two parts are operated in parallel for implementing pipelining concept and to increase the execution speed.

1.3.1. Bus interface unit (BIU) :

The BIU handles all transfer of address, data and code on Buses. The BIU consists of

(i) Instruction Queue
(ii) Segment Registers
(iii) Instruction Pointer
(iv) 20-bit Physical address calculation circuit

(i) Instruction Queue:

The pipelining concept is implemented in 8086 using an Instruction Queue. While Execution Unit (EU) is executing an instruction, the BIU will fetch upto 6- instruction code bytes from memory and stores them in a FIFO group of registers called as Instruction Queue. When EU is ready for its next instruction, it simply takes instruction bytes from Instruction Queue in BIU. This is much faster than sending out an address to memory, waiting for memory to send-back the next instruction bytes.

Note that the Instruction Queue fails in the case of branch instructions execution (Ex: JMP, CALL, RET, etc). After the execution of branch instructions, the program execution jumps to different location. Hence, the instructions available in the Instruction Queue will be erased during the execution of branch instructions.

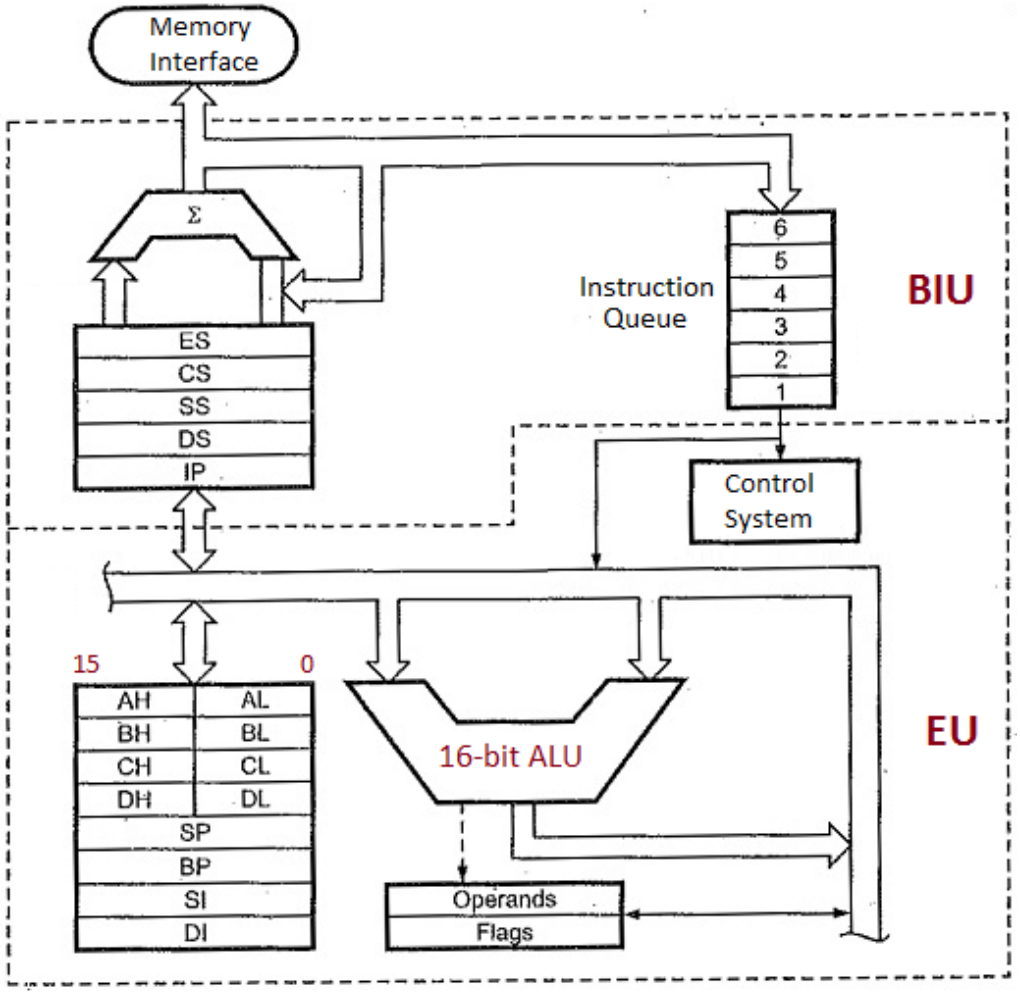


Figure 1.1. Internal architecture and registers of 8086

(ii) Segment Registers

The 8086 uses memory segmentation. In this scheme, the complete 1 MB physical memory is divided into number of logical segments. The size of each segment is 64 KB in size and is addressed by one of the segment registers. The segment registers in BIU are used to define the starting address of logical segments.

The four segment registers in 8086 are

Code segment register (CS) → defines the starting address of Code segment
Data segment register (DS) → defines the starting address of Data segment
Extra segment register (ES) → defines the starting address of Extra segment
Stack segment register (SS) → defines the starting address of Stack segment

The segment registers in BIU hold the upper 16-bits of starting address of logical segments. The BIU inserts ZERO's for lower 4-bits of 20-bit starting address.

For example, if CS=2000 H, then the Code segment starts at 20000H.

(iii) Instruction Pointer (IP) :

The IP register holds the ***offset address*** of the next instruction to be fetched from Code segment. The Offset address is defined as the distance of operand from the Starting address of the Segment.

(iv) 20-bit Physical address calculation circuit

The BIU has a Physical Address Generation Circuit.
It generates the 20-bit physical address by adding Segment base to the Offset address.

The segment register defines the segment base address and a location within a segment can be addressed by *16-bit offset address*.

For example, if CS= 2000H and IP= 5678H then

Segment base = CS*10 = 20000 H
Offset address = IP = 5678 H

20-bit Physical address = Segment base + Offset = 25678 H

Note that the 20-bit Physical address is generally represented in the form of

Segment base : Offset = CS:IP = 2000:5678 H

1.2.2. Execution Unit (EU) :

The Execution unit is responsible for decoding and executing the instructions. It receives instruction code bytes from Instruction Queue in BIU, decodes and executes them.

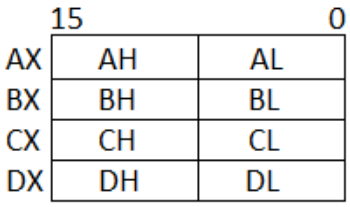
The EU consists of

(i) 16-bit ALU
(ii) General purpose registers: AX,BX,CX,DX
(iii) Index Registers : SI and DI
(iv) Pointers : SP and BP
(v) Flag Register/Program Status Word (PSW)

(i) 16-bit ALU :

The EU consists of 16-bit ALU which can perform Arithmetic operations (such as Addition, Subtraction, Multiplication, Division, Increment, Decrement, etc) and Logic operations (such as Logic AND, OR, NOT, Ex-OR, Shift, Rotate, etc).

(ii) General purpose registers :



The 8086 has 4- general purpose registers - AX, BX, CX, DX.
All these registers are 16-bit registers used to store 16-bit data
Each register is divided in two 8-bit registers to store 8-bit data
These registers have multiple functions, shown in following Table.

Register	*Name*	*Purpose*
AX	Accumulator	Used to hold the results of some operations like MUL, DIV, Shift, Rotate…etc
BX	Base Index	Used to hold the offset address (Based addressing)
CX	Counter	Used as a counter in LOOP and String instructions
DX	Data Index (or) Extended Accumulator	Used to hold a part of result from MUL & DIV. It is also used to hold the 16-bit I/O port address during I/O operation.

(iii) Index Registers :

- The Index Registers are used to hold the 16-bit offset address of data stored in Data and Extra segments.
- These registers are used in string operations to hold the offset address of source string and destination strings.
  The SI register holds the offset address of source string in Data Segment.
  The DI register holds the offset address of destination string in Extra Segment.
  - Address of source string → DS:[SI]
  - Address of destination string → ES:[DI]

- The directional flag (DF) selects the increment (or) decrement mode for SI and DI registers during String instructions manipulation.
  DF = 0 selects increment mode and
  DF = 1 selects decrement mode.

(iv) Pointers :

- The Pointers are used to hold the offset address relative to data and stack segments.
- The base pointer (BP) is used to access data in stack segment.
- The stack pointer (SP) is used to hold the address of stack top.

Register	*Name*	*Purpose*
SI	Source Index	It is used in string operations to hold the offset address of source string in Data segment.
DI	Destination Index	It is used in string operations to hold the offset address of destination string in Extra segment.
BP	Base Pointer	It is used to access data the stack segment. It is used to hold the offset address (Based addressing)
SP	Stack Pointer	It is used to hold the address of Stack top

(v) Flag Register / Program Status Word (PSW)

The flag register is used to indicate the status information (or) condition produced by an instruction execution.
The 8086 has 6- status flags and 3-control flags.

Conditional Flags / Status Flags :

These flags are Set or Reset according to the condition produced by an instruction execution.
The 6- conditional flags of 8086 are CF, PF, AF, ZF, SF, OF

Control flags:

These flags control the operation of the processor.
The 3- control flags of 8086 are DF, IF, TF.

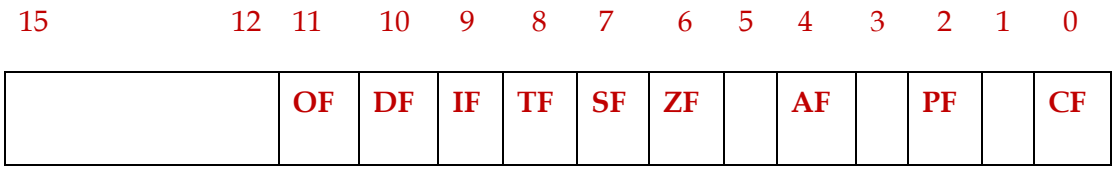


Figure: Flag register of 8086

Carry Flag (CF) : It is set to 1, if a carry is generated in Addition (or) Subtraction

Parity Flag(PF) : It is set to 1, if the result of an operation has even number of 1's

Auxiliary carry Flag (AF) : It is used in BCD operations.
It is set to 1, if addition of lower nibble generates a carry.

Zero Flag (ZF) : It is set to 1, if the result of an operation is ZERO

Sign Flag (SF) : It is used with signed numbers only.
SF=1 for –ve results and SF =0 for +ve results

Overflow Flag (OF) : It is set to 1, if the result of a signed operation exceeds the register capacity.

Directional Flag (DF) :

It is used in string operations to select either increment (or) decrement mode for SI and DI registers. DF = 0 selects increment mode and
DF = 1 selects decrement mode.

Interrupt Flag (IF) :

- If IF = 1, all the maskable interrupts are recognized and processed by the CPU.
- Otherwise, the maskable interrupts are ignored and not processed by CPU.

Trap Flag (TF) :

- It is used for debugging purpose
- If TF = 1, the processor enters the 'Single step execution mode' i.e., the 8086 gets interrupted automatically at the end of every instruction execution.

Problem :

The content of registers AL and BL are 79H and 88H respectively. Determine the status of control flags of 8086, after the execution of the instruction ADD AL,BL?

Ans:

```
AL = 79 H  =   0 1 1 1  1 0 0 1      ← AL
BL = 88 H  =   1 0 0 0  1 0 0 0      ← BL
-----------------------------------------------
ADD AL, BL  :   0 0 0 0  0 0 0 1     ← AL
-----------------------------------------------
```

The status of conditional flags are :

CF = 1 ;
PF = 0;
AF = 1 ;
ZF = 0 ;
SF = 0 ;
OF = 0

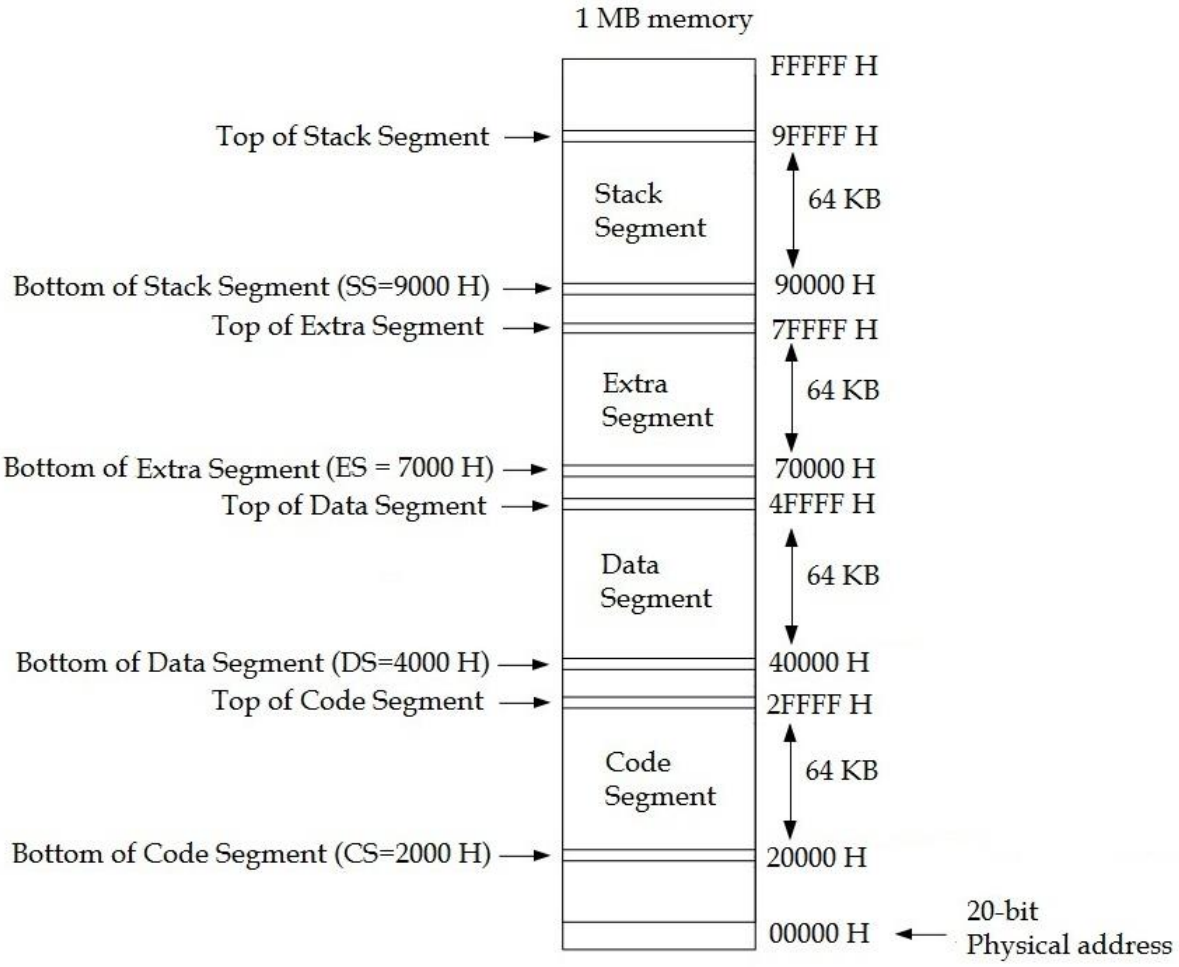
1.3. MEMORY SEGMENTATION

The 8086 uses memory segmentation. In this scheme, the complete 1 MB physical memory is divided into number of logical segments. The size of each segment is 64 KB in size and is addressed by one of the segment registers.

The 64 KB logical segment can be located anywhere in 1 MB memory, but the segment will always start at an address with ZERO's in lowest 4-bits.

Note that the 8086 does not work the whole 1 MB memory at any given time. However it works only with four 64 KB segments within the whole 1MB memory.

The four segment registers in BIU define the starting addresses of the four memory segments with which the 8086 is working at that instant of time. The segment registers in BIU are used to hold the upper 16-bits of starting address of logical segments. The BIU inserts ZERO's for lower 4-bits of 20-bit starting address.



Code Segment :

- The Code segment is used to store the program instruction codes.
- The Code Segment register (CS) is used to hold the upper 16-bits of starting address of the Code segment. The BIU inserts ZERO's for lower 4-bits of 20-bit starting address.
- For example, if CS=2000H then the Code segment starts at 20000H.

Data Segment :

- The Data segment is used to store data variables and constants of the program.
- The Data Segment register (DS) is used to hold the upper 16-bits of starting address of the Data segment.
- Data are accessed from Data segment by an Offset address.
- The SI, DI, BX, BP registers are used to store the offset address for data segment

Extra Segment :

- The Extra segment is an additional data segment.
- It is used in String operations to store the destination string.
- The Extra Segment register (ES) is used to hold the upper 16-bits of starting address of the Extra segment.
- The SI, DI, BX, BP registers are used to store the offset address for data segment

Stack Segment :

- The Stack segment defines the area of memory used for stack.
- Stack is a section of memory where the data are accessed in LIFO manner
- The stack memory is used to store data, address and status information. It is used by CPU to store return address during the execution of procedures and ISRs.
- The Stack Segment register (SS) is used to hold the upper 16-bits of starting address of the Stack segment.
- The Stack Pointer (SP) register holds the address of Stack top.

Addressing a location within a segment (20-bit Physical address calculation) :

The BIU has a Physical Address Generation Circuit.
It generates the 20-bit physical address by adding Segment base to the Offset address.

The segment register defines the segment base address and
a location within a segment can be addressed by *16-bit offset address* as shown in Figure.

For example, if CS= 2000H and IP= 5678H then

Segment base = CS*10 = 20000 H
Offset address = IP = 5678 H

20-bit Physical address = Segment base + Offset = 25678 H

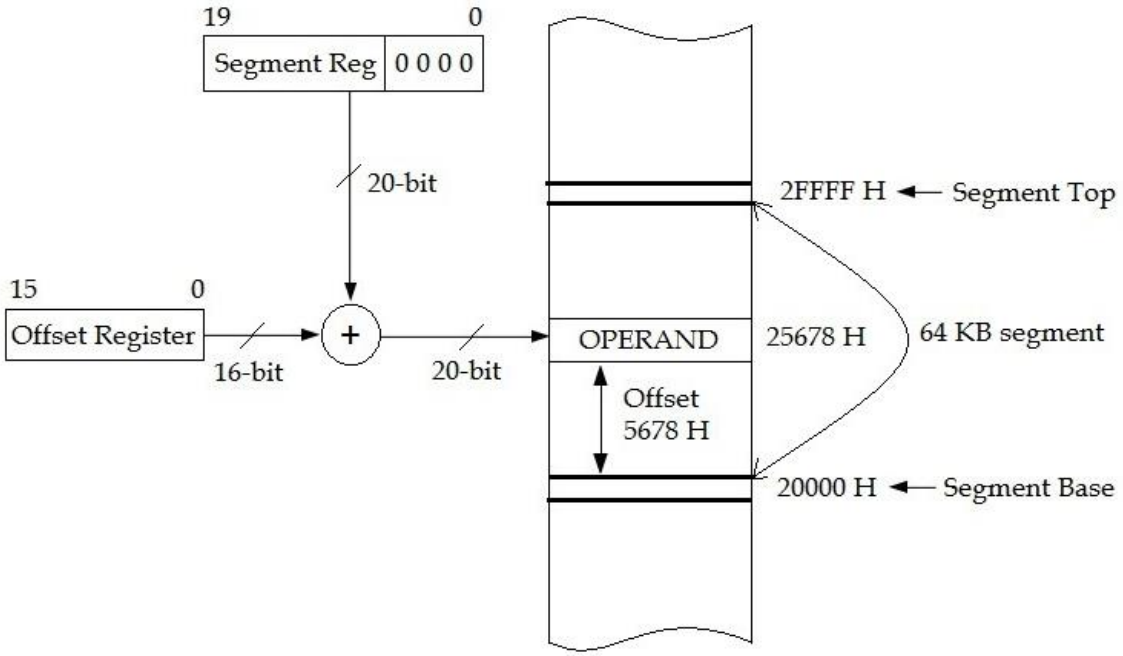


Figure : 20-bit Physical address calculation

The segment register holds the upper 16-bits of starting address of logical segments. The BIU inserts ZERO's for lower 4-bits of 20-bit starting address.

The Offset address is defined as the distance of operand from the Starting address of the Segment. The registers used to store the 16-bit offset addresses for various segments are given in Table.

Segment	*Segment Register*	*Offset Register*
Code Segment	CS	IP
Data Segment	DS	SI (DI/BX/BP)
Extra Segment	ES	DI (SI/BX/BP)
Stack Segment	SS	BP /SP

Advantages of Memory Segmentation :

- Allows to access 1 MB memory with 16-bit address.
- Allows Separate memory areas for program, data and stack
- Provides data and code protection
- Provision for relocation of programs and data
- Provides a powerful memory management mechanism
- Allows the processor to access the data from memory easily and fastly, which increases the speed of operation

1.4. REGISTER ORGANIZATION OF 8086

The Intel 8086 has a powerful set of registers as shown in figure.
The description of all registers is given in section 1.2

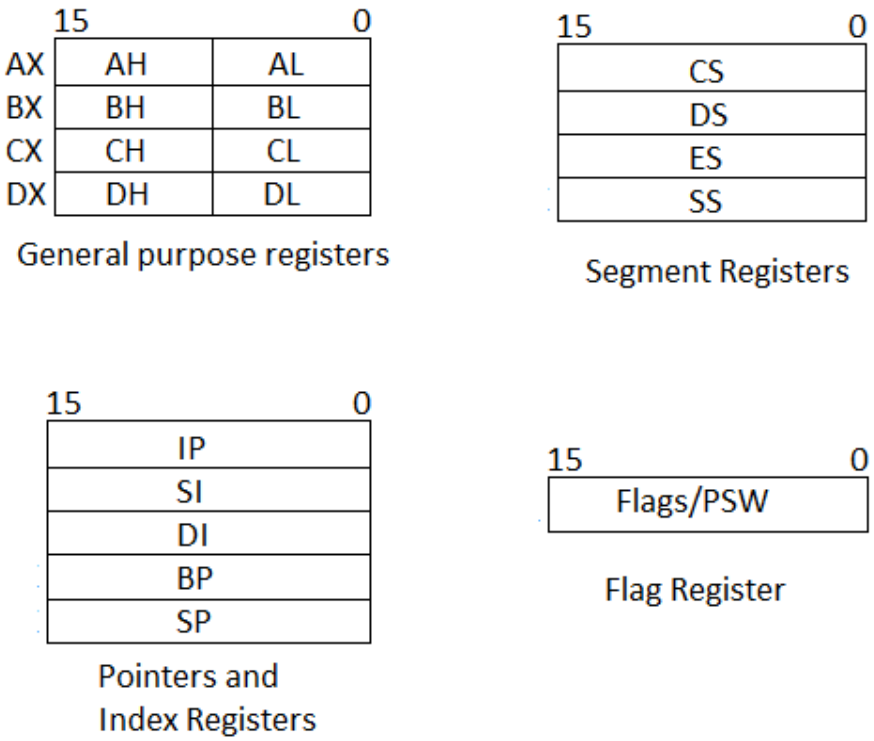


Figure: Register organisation of 8086

1.5. PIN CONFIGURATION OF 8086 :

The pin diagram of 8086 is shown in Figure.
The 8086 consists of 40-pins.
Among these, 21 are multiplexed pins to reduce the number of pins

$AD_0 - AD_{15}$
$A_{16}/S_3 - A_{19}/S_6$ and
$\overline{BHE}$ / S_7

The 8086 issues two different sets of signals (Pins 24 to 31) in Min. mode and Max. mode.

Min. Mode → $\overline{INTA}$, ALE, $\overline{DEN}$, DT/$\overline{R}$, M/$\overline{IO}$, $\overline{WR}$, HLDA, HOLD

Max. Mode → QS1, QS0, $\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$, $\overline{WR}$, RQ/$\overline{GT_1}$, RQ/$\overline{GT_0}$

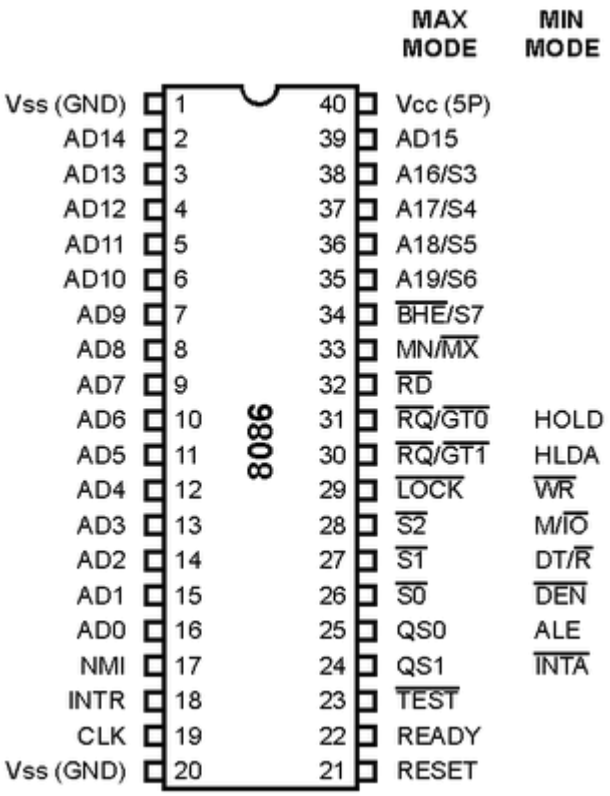


Figure: Pin diagram of 8086

AD_0 - AD_{15} (Address / Data lines) :

- The lower order 16-address lines (A_0 - A_{15}) of 8086 are multiplexed with 16- data lines (D_0 - D_{15}).
- During T_1 state of Bus-cycle, the 8086 sends out address on these lines and during later part of Bus-cycle this multiplexed bus is used as Data bus.
- All the above multiplexed pins are at high-impedance state during DMA operation.

A_{16}/ S_3 - A_{19}/ S_6 (Address/Status lines) :

- The higher order 4-address lines (A_{16} - A_{19}) are multiplexed with Status lines (S_3 - S_6).
- The **S_3 and S_4** shows which segment is accessed during current bus cycle.
- The S_5 indicates the status of IF flag.
- The S_6 is always logic 0 and it is not used.
- All the above multiplexed pins are at high-impedance state during DMA operation.

S4 S3	*Segment*
0 0	Code Segment
0 1	Data Segment
1 0	No Segment
1 1	Extra Segment

$\overline{BHE}$ / S_7 :

- The 1 MB Physical memory of 8086 is divided into two banks **for accessing 16-bit numbers**. Each bank size is 512 K bytes.
- The Bus High Enable $\overline{BHE}$ signal is used to enable the higher order data bus (D_8 - D_{15}) connected to HIGH BANK.
- The status signal S_7 is used by arithmetic co-processor 8087 to determine whether the CPU is 8086 (or) 8088.

$\overline{BHE}$	A_0	*Bank Selected*
0	0	Both banks are enabled for 16-bit operation
0	1	HIGH bank is enabled
1	0	LOW bank is enabled
1	1	No banks are enabled

NMI : Non-Maskable Interrupt:

- It is a positive going **edge triggered** Non-Maskable Interrupt
- It cannot be disabled by using software.
- This interrupt has highest priority than INTR.
- It is a vectored interrupt and the vector address of NMI is 0000:0008 H.

.

INTR: Interrupt Request :

- It is a **level triggered** maskable Interrupt Request.
- It can be disabled by using software i.e., the processor will get interrupted only if IF=1. Otherwise the processor will not get interrupted even INTR is active.
- It is a non-vectored interrupt.

CLK:

- The clock input provides the basic timing for µp and bus control activity.
- The clock frequencies of different versions of 8086 are 5 MHz, 10 MHz, 8 MHz.

RESET :

- It forces all the registers to a predefined values and microprocessor gets reset.
- When Reset is active DS, ES, SS, IP and FLAG registers are initialized to 0000H and CS is initialized to FFFF H.
- After Reset, the processor starts execution from FFFF0 H.

READY:

- A slow peripheral (or) memory device can be connected to microprocessor through READY line. It is used to insert wait states in bus cycles as needed to interface with slow memory & I/O
- It is used by the MPU to sense whether the peripheral is ready to transfer data or not

  If READY =1, peripheral is ready to transfer data.

  If READY=0, the processor WAITS until it goes to HIGH

$\overline{\text{TEST}}$:

- This input is examined by WAIT instruction.
- The 8086 enters into WAIT state after the execution of WAIT instruction.

  If $\overline{\text{TEST}}$ = 0, the WAIT instruction functions as NOP

  If $\overline{\text{TEST}}$ =1, the processor waits until $\overline{\text{TEST}}$ = 0.
- If the co-processor has finished its work then it makes $\overline{\text{TEST}}$= 0.

$\overline{\text{RD}}$:

The **Read Control Signal** is active whenever the processor is ready to read data from Memory (or) I/O.

MN/$\overline{\text{MX}}$:

OPERATING MODES OF 8086/8088:

- The 8086 can be operated in two modes – Minimum mode and Maximum mode.
- The pin MN/$\overline{\text{MX}}$ is used to select the Min. (or) Max. mode of operation.

  MN/$\overline{\text{MX}}$ = 1 for Minimum mode operation

  MN/$\overline{\text{MX}}$ = 0 for Maximum mode operation

  - The pins (24-31) issues two different set of signals – for minimum & maximum mode operations. *These pins are described below:*

Note : The Min.mode pins and Max.mode pins are described in sections 1.6.and 1.7.

Minimum mode:

- The min. mode operation is selected by connecting the pin MN/MX to + 5 V
- The 8086 is operated in minimum mode in simple systems with a single CPU
- In min. mode operation, all the control signals are generated by CPU
- It is least expensive way to operate and the operation is similar to 8085A.
- Min. Mode signals → $\overline{\text{INTA}}$, ALE, $\overline{\text{DEN}}$, DT/$\overline{\text{R}}$, M/$\overline{\text{IO}}$, $\overline{\text{WR}}$, HLDA, HOLD

Maximum mode:

- The max. mode operation is selected by connecting the pin MN/MX to GND
- The 8086 is operated in maximum mode in multi-processor system with more than one processor.
- In max. mode, the control signals are generated by external Bus-controller 8288.
- The maximum mode operation is used only when the system contains arithmetic co-processor such as 8087 numeric co-processor.
- Max. Mode signals → QS_1, QS_0, $\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$, $\overline{\text{LOCK}}$, RQ/$\overline{GT_1}$, RQ/$\overline{GT_0}$

1.6. MINIMUM MODE CONFIGURATION OF 8086 :

- The min. mode operation is selected by connecting the pin MN/MX to + 5 V
- The 8086 is operated in minimum mode in simple systems with a single CPU
- In min. mode operation, all the control signals are generated by CPU
- It is least expensive way to operate and the operation is similar to 8085A.
- Min. Mode signals → $\overline{INTA}$, ALE, $\overline{DEN}$, DT/$\overline{R}$, M/$\overline{IO}$, $\overline{WR}$, HLDA, HOLD

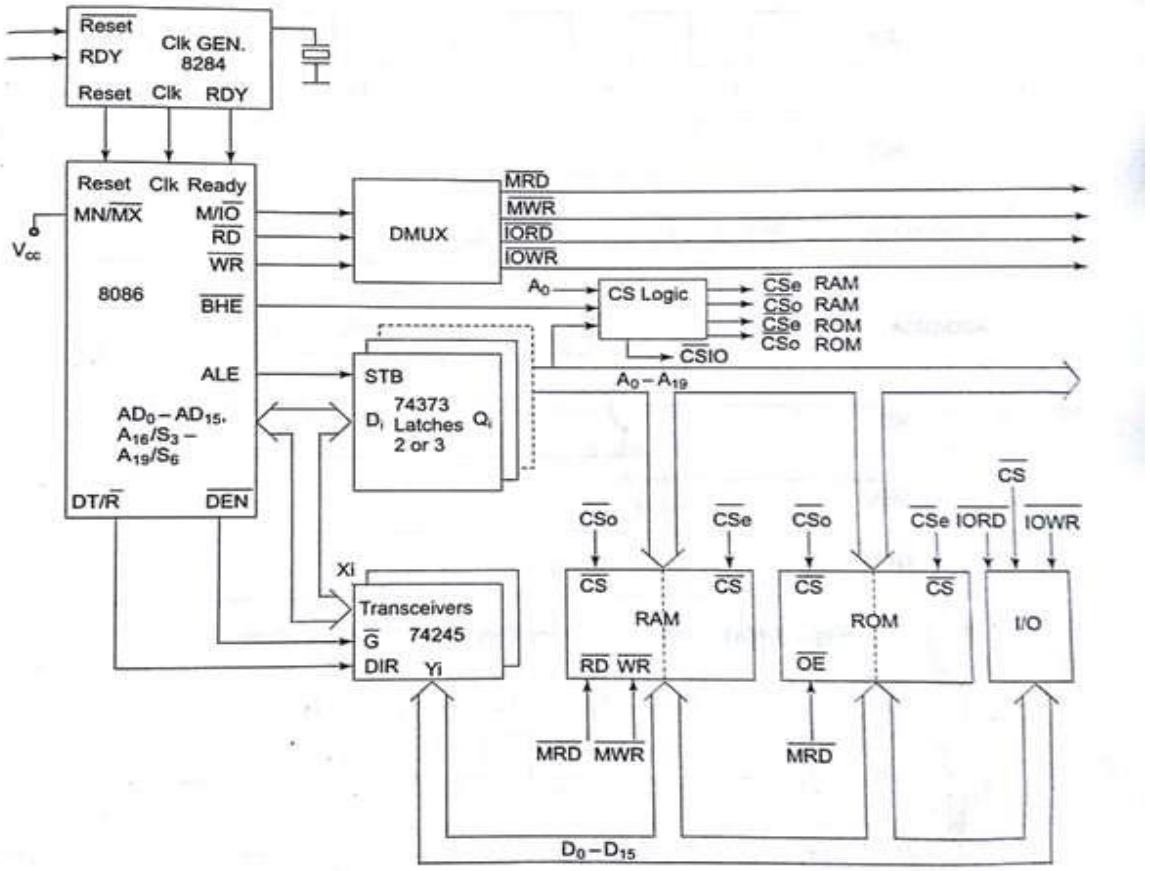


Fig.1.13 *Minimum Mode 8086 System*

Clock generator :

- It generates a CLK of frequency 5 MHz
- It provides the basic timing for µp and bus control activity
- It also synchronizes some external signals with the system clock

Address Latches :

- Latches are used for de-multiplexing of $AD_0 - AD_{15}$, A_{16}/S_3 - A_{19}/S_6 and $\overline{BHE}$ / S_7
- Latches are used for the separation of address and data lines
- Address latch enable (ALE) signal is used to enable the Latches

Transceivers/ Bi-directional data buffers

- The Bi-directional data buffers (transmitters/receivers) are used to maintain proper signal quality. i.e., to increase the fan-out of the system.
- The $\overline{DEN}$ signal is used to enable the bi-directional data buffers
- The DT/$\overline{R}$ signal controls the flow of data through data buffers

MINIMUM MODE SIGNALS:

$\overline{\text{INTA}}$ (pin-24) :

- The interrupt acknowledge signal is a response to the INTR.
- It indicates the recognition of an Interrupt Request.

ALE (pin-25) :

- The address latch enable signal is used to indicate the presence of valid address information on multiplexed bus (AD_0 - AD_{15})
- This signal is used to enable the Address Latches.
- The address latches are used to separate the address and data lines.(demultiplexing)

$\overline{\text{DEN}}$ (pin-26) :

- The data buffers enable signal is used to enable the bi-directional data bus buffers.
- The data bus buffers or transceivers (transmitters/receivers) are used to maintain proper signal quality.

DT/$\overline{\text{R}}$ (pin-27) :

- The data transmit / receive signal is used to indicates the direction of data flow through the data bus buffers.

  DT/$\overline{\text{R}}$ = 1 for transmitting data

  DT/$\overline{\text{R}}$ = 0 for receiving data

M/$\overline{\text{IO}}$ (pin-28) :

It selects either Memory or I/O operation

M/$\overline{\text{IO}}$= 1 selects memory operation

M/$\overline{\text{IO}}$= 0 selects I/O operation

M/$\overline{\text{IO}}$	$\overline{\text{RD}}$	$\overline{\text{WR}}$	Operation
1	0	1	Memory Read
1	1	0	Memory Write
0	0	1	I/O Read
1	1	0	I/O Write

$\overline{\text{WR}}$ (pin-29) :

This control signal is active whenever the processor is writing data to Memory or I/O

HOLD & HLDA (pins 31 & 30):

- These two signals are used in DMA operation
- The HOLD input indicates the processor that other bus master (DMA controller) is requesting for the use of system bus.
- When HOLD =1, the µp stops the normal program execution and places address, data & control buses at high-impedance state and sends HLDA signal to the DMA controller.
- The HLDA signal indicates that the processor has accepted the HOLD request and the gain control of the system bus is transferred to the DMA controller.
- During HLDA =1 , the DMA controller is the master of the system bus.
- After removal of HOLD request, the HLDA becomes Low.

Timing diagrams for Min. mode

- The working of microprocessor based system can be explained with the help of Timing diagrams.
- The timing diagrams provide the information about the various conditions of the signal while a machine cycle is executed.

T- State : It is one cycle of the clock (clock period)

Machine cycle / Bus cycle :

- The group of T-States required for a basic bus operation is called as Machine cycle
- The microprocessor uses bus cycles for memory and I/O operations

  Opcode fetch cycle
  Memory Read cycle
  Memory Write cycle
  I/O Read cycle
  I/O Write cycle

Instruction cycle:

- The time required for the microprocessor to fetch, decode & execute an instruction is called an Instruction cycle.
- An instruction cycle consists of one (or) more bus cycles.

The timing diagram for Memory Read cycle in Minimum mode is shown in figure.

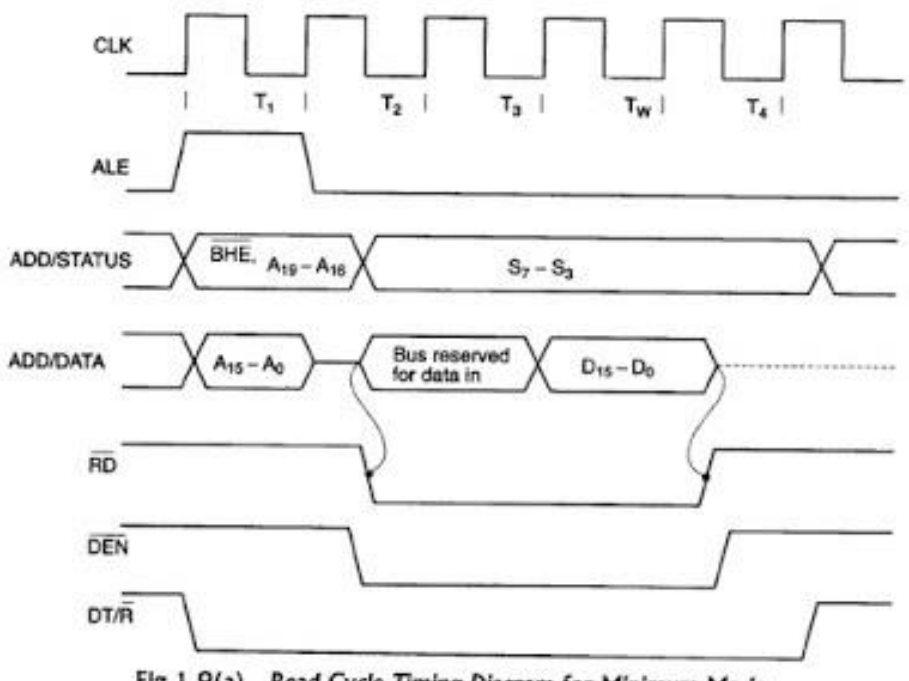


Fig. 1.9(a) *Read Cycle Timing Diagram for Minimum Mode*

During the time period T_1

- The micro processor sends out 20 bit address.
- Enables ALE Signal
- Issues $M/\overline{IO} = 1$ for memory operation
- Issues $DT/\overline{R} = 0$ for data reception

During the time period T_2

- The address information is removed from multiplexed bus and status signals are placed. However A0-A19 remain available as they were latched during T_1
- Enables $\overline{RD}$ signal
- Enables $\overline{DEN}$ signal

During the time period T_3

- The microprocessor checks the READY line. If READY =1, the μP reads the data from data bus. Otherwise, a WAIT state T_W is inserted

During the time period T_4

- All the control signals are deactivated to start the next cycle.
- $\overline{DEN}$, $DT/\overline{R}$, $M/\overline{IO}$ and $\overline{RD}$ signals are deactivated

1.7. MAXIMUM MODE CONFIGURATION OF 8086 :

- The max. mode operation is selected by connecting the pin MN/MX to GND
- The 8086 is operated in max. mode in multi-processor system with more than one processor.
- **In max. mode, the control signals are generated by external Bus-controller 8288.**
- The maximum mode operation is used only when the system contains arithmetic co-processor such as 8087 numeric co-processor.
- Max. Mode signals → QS_1, QS_0, $\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$, $\overline{LOCK}$, $RQ/\overline{GT_1}$, $RQ/\overline{GT_0}$

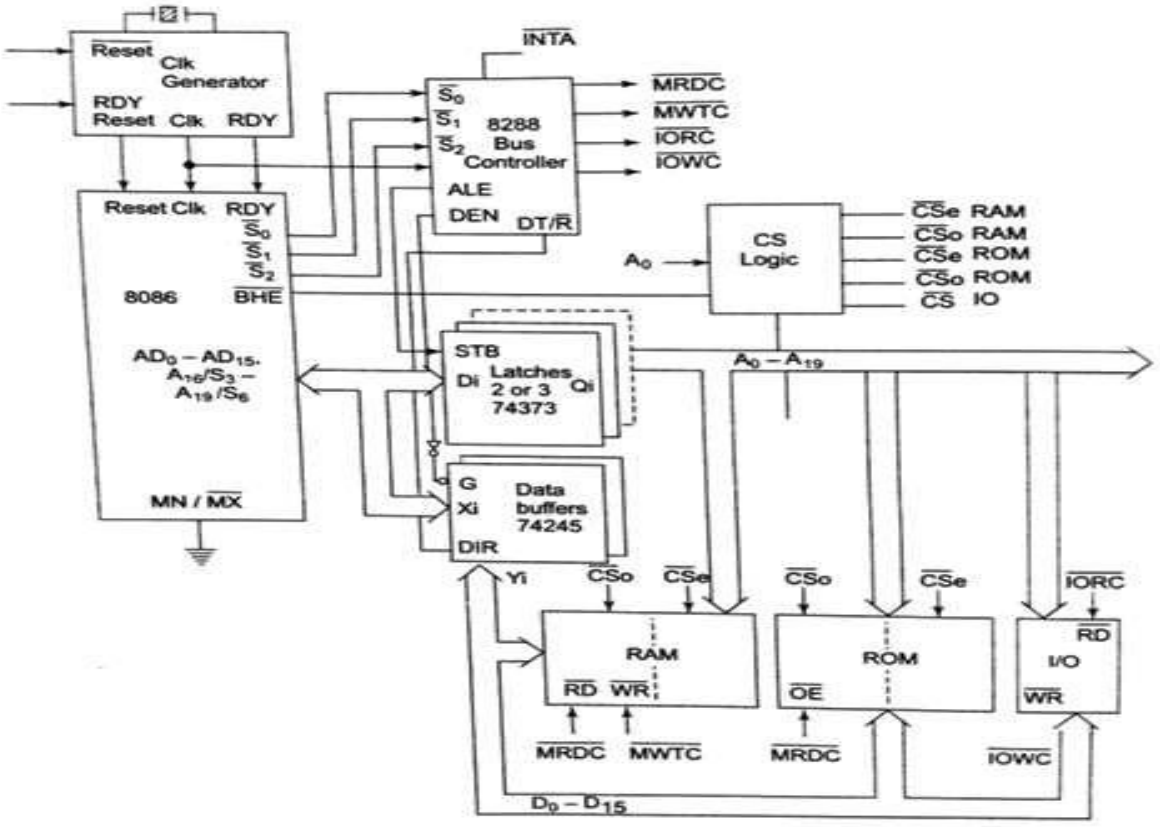


Fig. 1.15 *Maximum Mode 8086 System*

MAXIMUM MODE SIGNALS:

QS_1 & QS_0 (pins-24&25):

The Queue status pins indicate the status of Instruction Queue of 8086 processor.

QS_1	QS_0	Queue Status
0	0	No operation
0	1	First byte of Opcode
1	0	Queue is empty
1	1	Next byte of Opcode

$\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$ (pins- 26,27&28):

These status signals indicate the function of the current bus-cycle. i.e., the type of operation being carried out by the processor.

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Bus cycle
0	0	0	INTA
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	HALT
1	0	0	Op-code fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	INACTIVE

$\overline{LOCK}$: (pin-29)

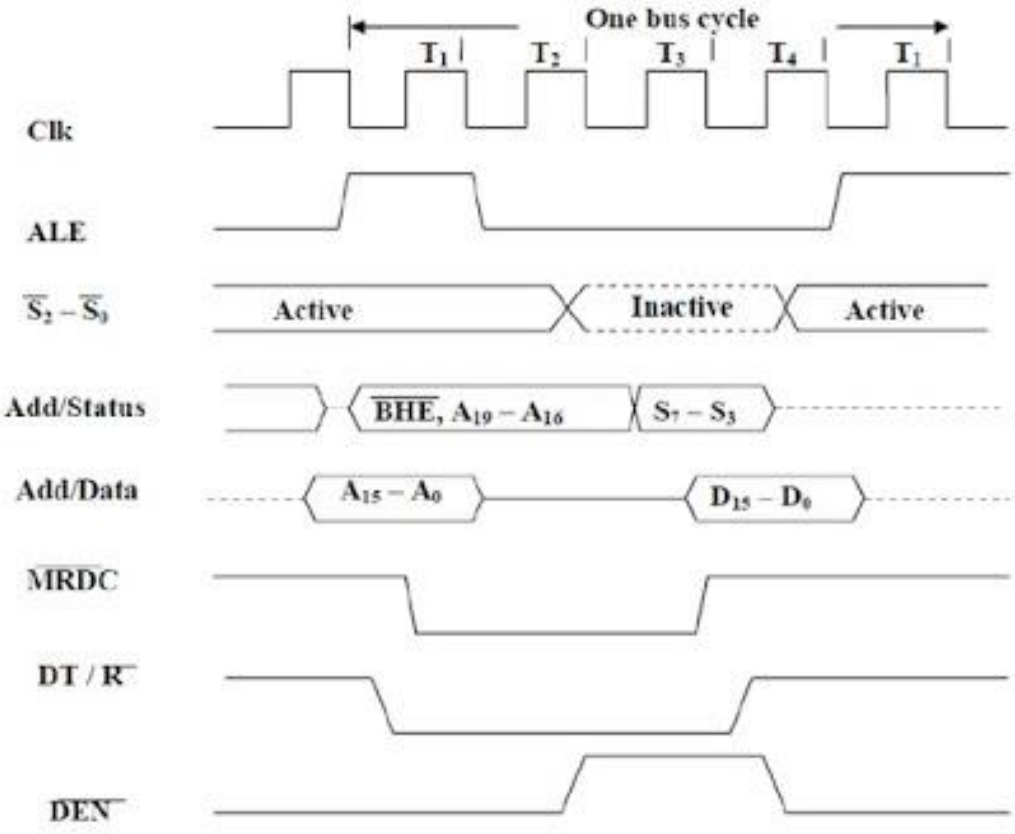
- This pin indicates that the processor is executing a LOCK prefixed instruction and the System bus is not to be used by another bus-master.
- This pin is used in multi-processor system to prevent other Bus masters from taking the control of System bus during the execution of a critical instruction.

$RQ/\overline{GT_1}$, $RQ/\overline{GT_0}$ (pins -30 & 31) :

- The Request/Grant pins are used to request a DMA action during Maximum mode operation.
- These are bidirectional and used to request and grant the DMA operation.

Timing diagrams for Max. mode

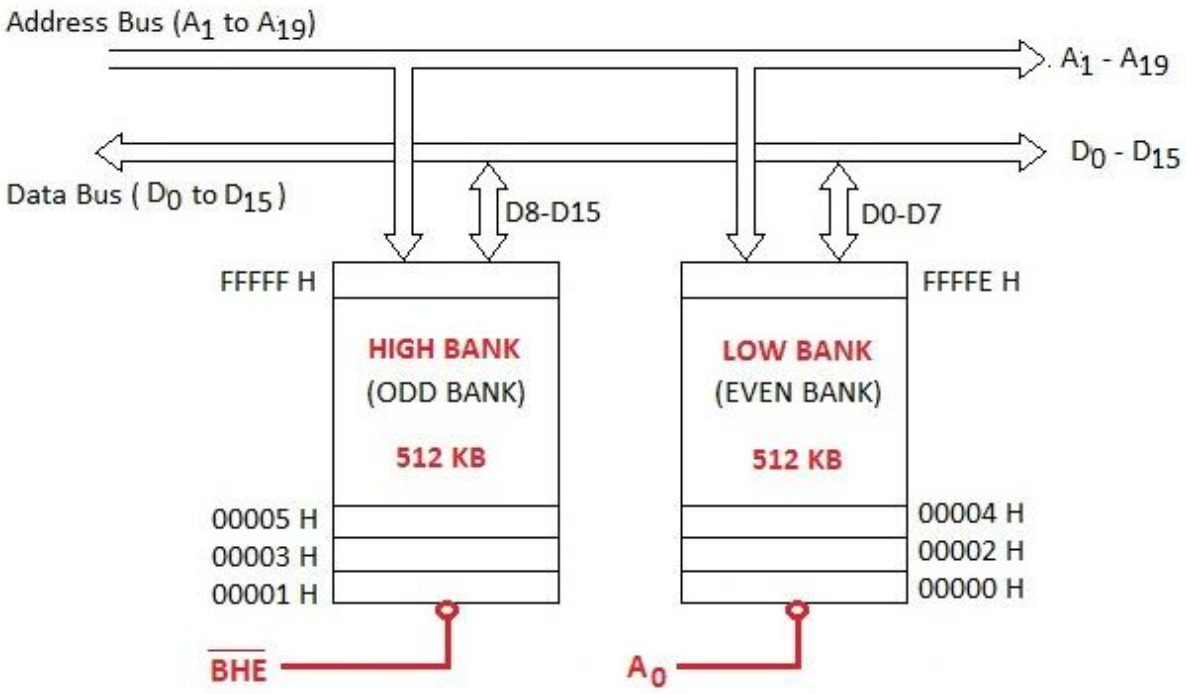
Op-code fetch cycle
Memory Read cycle
Memory Write cycle
I/O Read cycle
I/O Write cycle



Memory Read Timing in Maximum Mode

1.8. PHYSICAL MEMORY ORGANIZATION AND MEMORY BANKS ACCESSING

- The Physical memory of 8086 is divided into TWO banks **for accessing 16-bit numbers**. Each bank size is 512 K bytes.
- The data bus (D_0 - D_7) is connected to LOW bank /EVEN bank.
- The data bus (D_8 - D_{15}) is connected to HIGH bank/ ODD bank.
- Both Banks are enabled at a time for 16-bit operations



- Address lines (**A_1-A_{19}**) are used to select a location within the bank
- The LOW bank is selected by A_0
- The HIGH bank is selected by $\overline{BHE}$ signal.
- Both Banks are enabled at a time for 16-bit operations.
- Note that the address must be EVEN for 16-bit access.

$\overline{BHE}$	A_0	*Bank Selected*
0	0	Both banks are enabled for 16-bit operation
0	1	HIGH bank is enabled
1	0	LOW bank is enabled
1	1	No banks are enabled

1.9. INTERRUPTS OF 8086 & INTERRUPT VECTOR TABLE

- Interrupt is an event that causes the µp to stop the normal program execution.
- The µp services the interrupt by executing a subroutine called Interrupt Service Routine
- After executing ISR, the control is transferred back again to the main program.

There are 3 sources of interrupts for 8086

Hardware Interrupts	→	signal applied to NMI, INTR pins
Software Interrupts	→	by executing INT instructions
Error in program execution	→	Divide by Zero, Overflow, etc

Hardware Interrupts :

These interrupts occur as signals on the external pins of the microprocessor. 8086 has two pins to accept hardware interrupts, NMI and INTR.

NMI (Non Maskable Interrupt)

- It is a positive going edge triggered Non-Maskable Interrupt
- It cannot be disabled by using software.
- This interrupt has highest priority than INTR.
- On receiving an interrupt on NMI line, the microprocessor executes INT 4
- The µp obtains the ISR address from location 2 x 4 = 00008H from the IVT

.

INTR (Interrupt Request)

- It is a **level triggered** maskable Interrupt Request.
- It can be disabled by using software i.e., the processor will get interrupted only if IF=1. Otherwise the processor will not get interrupted even INTR is active.
- It is masked by making IF = 0 by software through CLI instruction.
  It is unmasked by making IF = 1 by software through STI instruction.
- It is a non-vectored interrupt. On receiving an interrupt on INTR line, the µp issues INTA pulse to the interrupting device. Then the interrupting device sends the vector number 'N' to the processor. Now the µp multiplies N x 4 and goes to the corresponding location in the IVT to obtain the ISR address.

Software Interrupts :

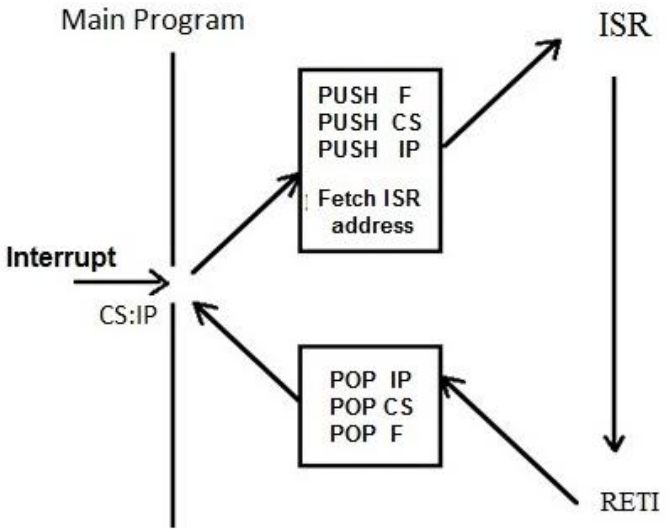
These interrupts are caused by writing the software interrupt instruction **INT N** where 'N' can be any value from 0 to 255 (00H to FFH). Hence all 256 interrupts can be invoked by software.

Error conditions :

The 8086 is interrupted when some special conditions occur while executing certain instructions in the program.

Example: Divide error, Overflow etc

Processing of Interrupts



When an interrupt is enabled, the 8086 µp performs the following actions

- ✓ It completes the execution of current instruction
- ✓ It pushes the Flag register (PSW) on to the stack
- ✓ The contents of CS and IP are pushed to stack (**return address**)
- ✓ It issues an Interrupt acknowledge (INTA)
- ✓ It computes the vector address from the type of interrupt
- ✓ The IP & CS are loaded with ISR address, which is available in Interrupt Vector Table
- ✓ Then the control is transferred to ISR and starts to execute the interrupt service routine at that address.

The code to handle an interrupt is called an *interrupt handler* or *Interrupt Service Routine* (ISR). An interrupt service routine must always finish with the special instruction IRET *(return from interrupt)*, which performs the following actions.

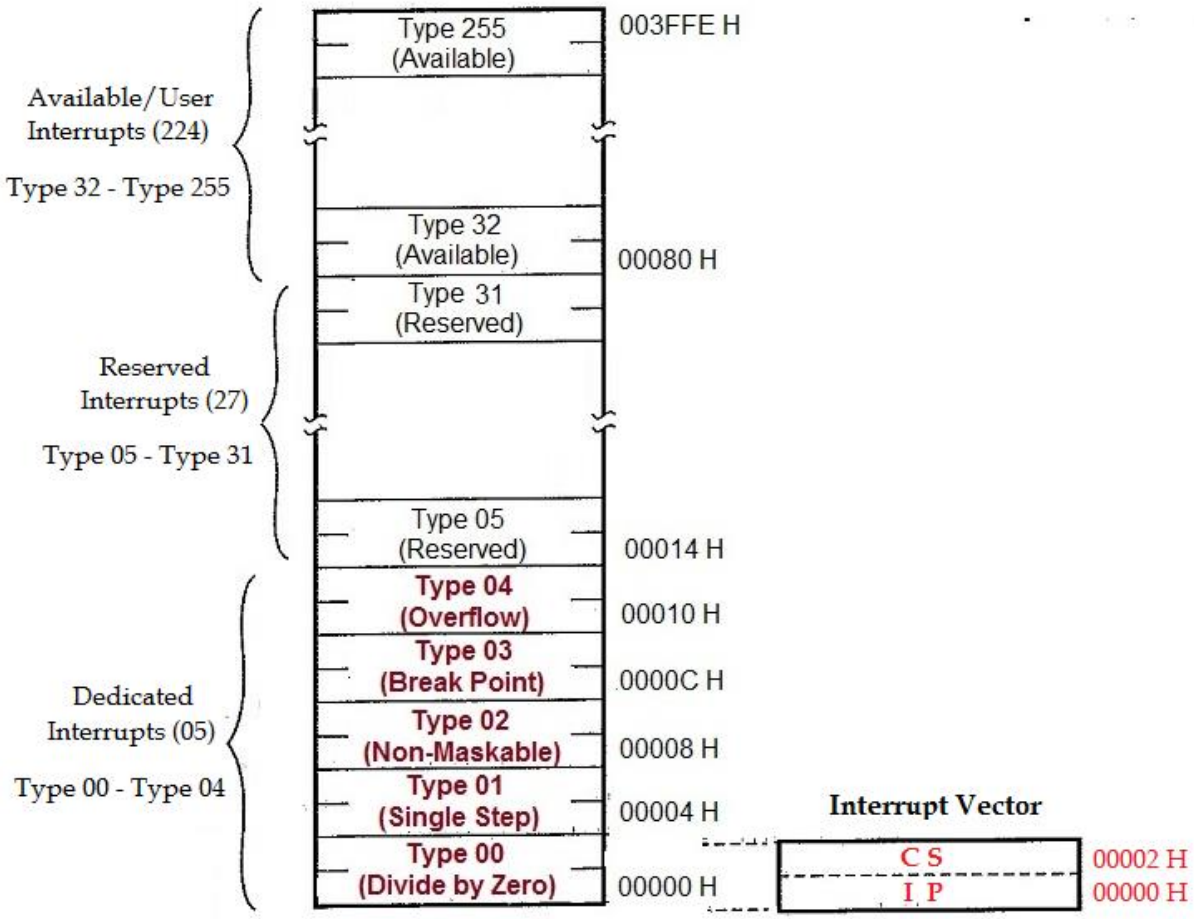
- ✓ The IP and CS are loaded with **return address** from stack
- ✓ The Flag register is popped from the stack
- ✓ The program execution returns to main line, where it was interrupted.

Interrupt Vector Table:

When the interrupt is enabled, the present IP and CS are pushed on to the stack and loaded with the address of ISR, which is available in Interrupt vector table.

Interrupt Vector → It is a memory block which contains address of ISR
The size of Interrupt vector is 4-bytes (to store CS and IP of ISR)

Interrupt Vector Table → It is a memory block which contains interrupt vectors
Since there are 256 types of interrupts, there are 256 interrupt vectors
Hence, the size of Interrupt Vector Table is 4*256 = 1024 bytes



8086 Interrupt Vector Table

In 8086 interrupt system, the first 1 KB memory from 00000 H to 003FF H is reserved for storing the starting addresses of ISRs. This block of memory is called as IVT.

- The Interrupt vector table contains 256 interrupt vectors
- **The interrupt vector contains IP and CS values of ISR**
- The address of interrupt vector can be calculated by multiplying the TYPE with 4
- *For example,* The interrupt vector address for Type-02 interrupt = 02 H*4= **00008 H**

- **The 8086 uses 256 types of interrupts – Type 00 to Type 255**
  These interrupts are classified as
  (A) Dedicated Interrupts (Type 00 to Type 04) : 05
  (B) Reserved Interrupts (Type 05 to Type 31) : 27
  (C) Available/User Interrupts (Type 32 to Type 255) : 224

(A) The dedicated interrupts (Type 0 – Type 4) are

(i) INT 0 : Divide Error
(ii) INT 1 : Single step execution (TRAP)
(iii) INT 2 : Non-Maskable Interrupt (NMI)
(iv) INT 3 : Break Point
(v) INT 4 : Overflow

(i) INT 0 (Divide Error)-

- This interrupt occurs whenever there is division error i.e. when the result of a division is too large to be stored. This condition normally occurs when the divisor is very small as compared to the dividend (or) the divisor is zero.
- Its ISR address is stored at location: Type 0 x 4 = **00000**H in the IVT.

(ii) INT 1 (Single Step)-

- The µp executes this interrupt after every instruction if the TF =1
- It puts µp in single stepping mode i.e. the microprocessor will get interrupted after executing every instruction. This is very useful during debugging.
- This ISR generally displays contents of all registers.
- Its ISR address is stored at location: Type 1 x 4 = **00004**H in the IVT.

(iii) INT 2 (Non Maskable Interrupt)-

- The microprocessor executes this ISR in response to an interrupt on the NMI (Non mask-able Interrupt) line.
- Its ISR address is stored at location: Type 2 x 4 = **00008**H in the IVT.

(iv) INT 3 (Breakpoint Interrupt)-

- This interrupt is used to cause breakpoints in the program.
- It is useful in debugging large programs
- This ISR generally displays contents of all registers
- Its ISR address is stored at location: Type 3 x 4 = **0000C**H in the IVT

(v) INT 4 (Overflow Interrupt)

- This interrupt occurs if the overflow flag is set
- It is used to detect overflow error in signed arithmetic operations.
- Its ISR address is stored at location: Type 4 x 4 = **0001**0H in the IVT

(B) Reserved interrupts (Type 5 to Type 31)

These levels are reserved by Intel to be used in higher processors like 80386, Pentium etc. They are not available to the user

(C) Available interrupts (Type 32 to Type 225)

- These are user defined, software interrupts.
- ISRs for these interrupts are written by the users to service various user defined conditions.
- These interrupts are invoked by writing the instruction INT N.
- Its ISR address is obtained by the microprocessor from location N x 4 in the IVT

Problem:

The contents of memory location 0000:008C are given below

0000:008C → 12, 34, 56, 78, 90, 92

(a) What is the interrupt vector address for Type-23H interrupt?

(b) Find the address of ISR corresponding to INT 23H

(c) ***For which type of interrupt, the interrupt vector address is 0000:00C8H***

Sol: (a) Interrupt vector address = Type * 4

For Type-23H interrupt, Interrupt vector address = 23*4 = 8C H

Interrupt vector address for Type-23H is 0000:008C H

(b) The address of ISR for Type-23H is available at 0000:008C H

ISR IP is available at memory 0000:008C H = 3412 H

ISR CS is available at memory 0000:008E H = 7856 H

Address of ISR = CS: IP = 7856:1234

(c) Interrupt vector address = Type * 4

00C8 = Type * 4

Type = 00C8 / 4 = 32H

Hence the Type of Interrupt is 32H

UNIT-2
ASSEMBLY LANGUAGE PROGRAMMING WITH 8086

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

2.1. ADDRESSING MODES OF 8086 :

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

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

The 8086 supports the following addressing modes:

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

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

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

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

Ex: MOV AX, BX
ADD AX, BX

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

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

4. Register Indirect addressing:

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

(a) Based addressing:

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

Ex: MOV AX, DS:[BX]

(b) Indexed addressing:

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

Ex: MOV AX, DS:[SI]

(c) Based Indexed addressing

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

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

5. Register Relative addressing:

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

(a) Relative Based addressing:

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

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

(b) Relative Indexed addressing:

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

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

(c) Relative Based Indexed addressing:

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

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

6. Implicit addressing:

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

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

7. I/O port addressing:

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

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

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

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

Ex: IN AL, DX
OUT DX, AL

8. Addressing modes for Control transfer / Branch Instructions

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

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

(a) Intra-segment mode :

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

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

Ex: JMP displacement
JMP SHORT Lable

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

(ii) Intra-segment Indirect :

In this mode, the destination address is found as content of Memory location.

destination IP = 16-bit Content of memory location

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

destination IP ← [BX]

(b) Inter segment mode :

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

(i) Inter segment Direct :

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

Ex: JMP 4000:6000
CALL 4000:6000

destination CS = 4000H,
destination IP = 6000H

(ii) Inter segment Indirect:

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

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

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

Problem :

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

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

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

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

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

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

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

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

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

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

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

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

Here, Source is AX
Destination is Stack

SP ← SP - 2
[SP] ← AX

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

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

Here, Source is Stack
Destination is CX

CX ← [SP]
SP ← SP + 2

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

2.2. INSTRUCTION FORMATS OF 8086 :

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

- Each instruction has 2- parts

Instruction

OPCODE	OPERAND

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

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

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

(i) One byte Instruction :

D7 DO

OPCODE

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

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

(ii) Register to Register :

D7 D0

OPCODE	W

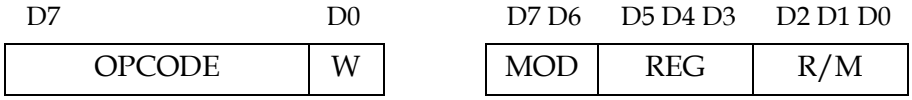
D7 D6 D5 D4 D3 D2 D1 D0

1 1	REG	R/M

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

Ex: MOV AX, BX

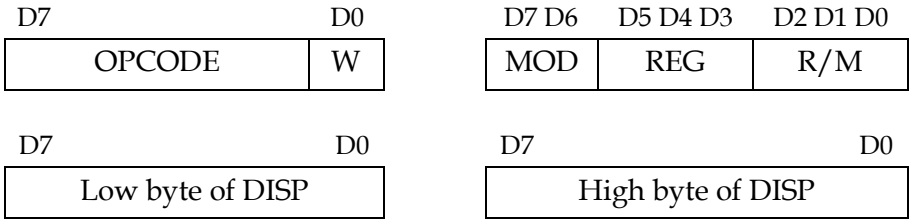
(iii) Register to/from Memory without Displacement



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

Ex: MOV AX, [SI]

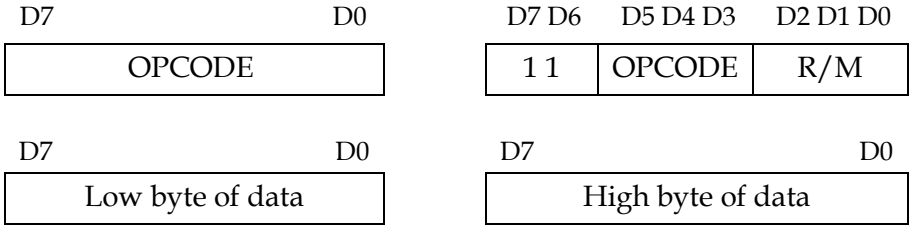
(iv) Register to/from Memory with Displacement



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

Ex: MOV AX, [SI+2000H]

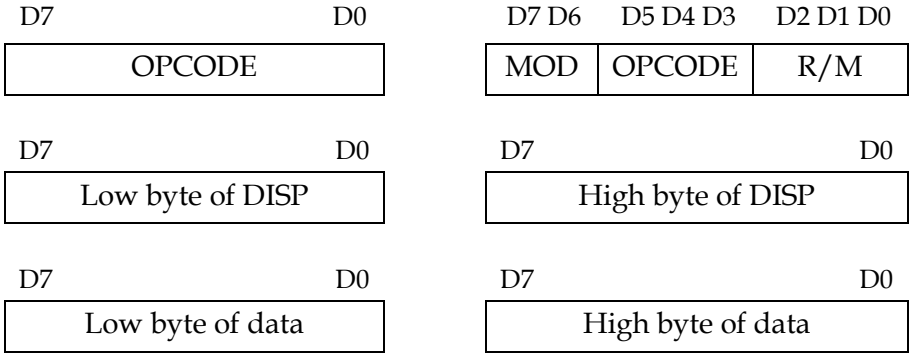
(v) Immediate operand to Register



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

Ex: MOV AX, 1234 H

(vi) Immediate operand to Mmeory with Displacement



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

Registers Codes

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

SREG	Segment Register
00	ES
01	CS
10	SS
11	DS

MOD, REG and R/M filed Codes

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

2.3. INSTRUCTION SET OF 8086 :

The 8086 Instruction set is classified as

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

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

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

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

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

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

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

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

2.4. ASSEMBLER DIRECTIVES :

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

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

1. Program Organization Directives

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

Syntax : Seg_name SEGMENT
Example : DATA SEGMENT

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

Syntax : Seg_name ENDS
Example : DATA ENDS

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

Syntax : END

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

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

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

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

2. Data definition and Storage Directives

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

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

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

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

```
Syntax    :    Var_name  DW value
Examples  :    N1  DW   1234 H
               ARRAY   DW  1000H, 2000H, 3000H, 4000H
```

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

```
Syntax    :    Var_name  DD value
Examples  :    N  DD   11223344 H
```

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

```
Syntax    :    Var_name  DQ value
Examples  :    N  DD   1122334455667788 H
```

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

```
Syntax    :    Var_name  DT value
Examples  :    N  DT   11223344556677889900 H
```

3. Alignment Directives

These directives are used to modify the memory location counter

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

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

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

```
Syntax      : EVEN
```

4. Procedure definition Directives

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

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

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

Syntax : Procedure_name ENDP
Example : Fact ENDP

5. Macro definition Directives

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

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

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

Syntax : ENDM

6. Value returning attribute Directives

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

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

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

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

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

7. Data control Directives

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

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

(ii) PUBLIC :

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

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

(iii) EXTRN :

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

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

Example:

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

PUBLIC  N1, N2

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

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

EXTRN   N1 : BYTE,   N2 : WORD

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

         MOV  AL, N1
         MOV  BX, N2

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

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

Example:

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

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

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

2.5. PROCEDURES

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

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

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

Two types of Procedures

(i) NEAR PROCEDURE
(ii) FAR PROCEDURE

(i) Near Procedure :

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

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

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

(i) Far Procedure :

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

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

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

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

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

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

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

PASSING PARAMETERS TO/FROM PROCEDURES :

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

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

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

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

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

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

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

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

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

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

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

2.6. MACROS

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

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

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

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

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

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

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

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

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

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

2.7. COMPARISION BETWEEN PROCEDURES & MACROS

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

2.8. EXAMPLES OF ASSEMBLY LANGUAGE PROGRAMS

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

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

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

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

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

down :  INC  SI
        INC  SI
        LOOP UP

        MOV BX, N
        DIV BX
        MOV  RES, AX
        HLT

CODE    ENDS
        END
```

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

```
CODE    SEGMENT
        ASSUME CS:CODE

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

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

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

CODE    ENDS
        END
```

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

```
CODE    SEGMENT
        ASSUME CS:CODE

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

UP:     MOV  AX, [SI]           ;  Read number

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

        MOV [DI],  AX           ; Store result

        INC SI
        INC SI
        INC  DI
        INC  DI
        LOOP UP
        HLT

CODE    ENDS
        END
```

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

```
CODE    SEGMENT
        ASSUME CS:CODE

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

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

        TEST AL, 01H
        JNZ  odd_num

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

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

down:   INC SI
        LOOP UP
        HLT

CODE    ENDS
        END
```

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

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

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

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

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

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

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

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

```
CODE    SEGMENT
        ASSUME CS:CODE

        MOV SI,  OFFSET  ARRAY
        MOV CX, N

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

UP:     MOV  AL, DS:[SI]

        TEST AL, 80H
        JNZ  odd_num

        INC BX
        JMP down

odd_num : INC DX

down:  INC SI

        LOOP UP
        HLT

CODE    ENDS
        END
```

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

```
CODE    SEGMENT
        ASSUME CS:CODE

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

UP:     CMP  AL, [SI]
        JNC  down

        MOV  AL, [SI]

down:  INC SI
        LOOP UP

        HLT
CODE    ENDS
        END
```

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

```
CODE    SEGMENT
        ASSUME CS:CODE

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

        MOV AL, 00H
        MOV BL, 01H

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

        HLT
CODE    ENDS
        END
```

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

```
CODE    SEGMENT
        ASSUME CS:CODE

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

        MOV AL, N                       ;  Number to be searched

UP:     CMP AL, [DI]
        JZ down

        INC  DI
        LOOP UP
        MOV AX,0000H
        JMP  exit

down: MOV AX,1111H

exit  : HLT

CODE    ENDS
        END
```

(9) Write an 8086 ALP to search a character 'R' in the given string.
If the character is found, store 1111H in AX. Otherwise store 0000H in AX

```
CODE    SEGMENT
        ASSUME CS:CODE

        MOV  DI,  OFFSET  STRING
        MOV CX,  SIZE  STRING      ;  Counter

        MOV AL,  'R'               ;  Character  to be searched

REPNE :  SCASB
         JE down

        MOV AX,0000H
        JMP  exit

down: MOV AX,1111H

exit   : HLT

CODE    ENDS
        END
```

(10) Write an 8086 ALP to count the number of characters 'R' in the given string.

```
CODE    SEGMENT
        ASSUME CS:CODE

        MOV  DI,  OFFSET  STRING
        MOV CX,  SIZE  STRING      ;  Counter

        MOV AL,  'R'               ;  Character  to be searched
        MOV BX, 0000H              ;  To store the count value

UP:     NOP
        SCASB
        JNE down

        INC BX                     ; increment BX only if 'R' is available

down: LOOP UP

        HLT

CODE    ENDS
        END
```

(11) Write an 8086 assembly language program to move a string of length of 8-bytes from 'OLD_HOME' to 'NEW_HOME'

```
CODE    SEGMENT
        ASSUME CS:CODE

        MOV  SI,  OFFSET  OLD_HOME
        MOV  DI, OFFSET  NEW_HOME
        MOV CX, N
        CLD

   REP: MOVSB

        HLT
CODE    ENDS
        END
```

(12) Write a Procedure to move a string from 'OLD_HOME' to 'NEW_HOME'

```
movstr  PROC  NEAR

        MOV  SI,  OFFSET  OLD_HOME
        MOV  DI, OFFSET  NEW_HOME
        MOV CX, N
        CLD

   REP: MOVSB
        RET

movstr  ENDP
```

(13) Write a Macro to move a string from 'OLD_HOME' to 'NEW_HOME'

```
MOVSTR  MACRO  OLD_HOME,  NEW_HOME,  N

        MOV  SI,  OFFSET  OLD_HOME
        MOV  DI, OFFSET  NEW_HOME
        MOV CX, N
        CLD

   REP: MOVSB

ENDM
```

(14) Write an Assembly language program to reverse a string of length 5 bytes

```
CODE    SEGMENT
        ASSUME CS:CODE

        MOV  SI,  OFFSET  ARRAY1      ; address of source string
        MOV  DI, OFFSET  ARRAY2       ;  address of result
        MOV CX, N                     ; length of the string
        CLD
        ADD DI, CX

UP:     MOV AL,  [SI]                 ; Read a byte from [SI]
        MOV [DI],  AL                 ; Store the byte at [DI]
        INC SI
        DEC  DI
        LOOP UP

        HLT
CODE    ENDS
        END
```

(15) Write a Program to compare two strings, which are located at ARRAY1 and ARRAY2 If two strings are equal than store 1111H in AX , otherwise store 0000H in AX

```
CODE    SEGMENT
        ASSUME CS:CODE

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

        MOV AX,1111H

REPE:   CMPSB
        JE down

        MOV AX,0000H                  ; if not equal only

down :  NOP
        HLT
CODE    ENDS
        END
```

(16) Write a Program to check whether the given string is Palindrome (or) not?

Combine – Reversing a string and Comparison of two strings

(17) Write an ALP to sort the given series of numbers in ascending order

```
CODE    SEGMENT
        ASSUME  CS:CODE

        MOV CX, N                   ;  counter

BACK :  MOV BX, N
        DEC BX
        MOV  SI, OFFSET ARRAY       ; address of array

  UP :  MOV AL,[SI]
        INC SI
        CMP AL,[SI]
        JB down

        XCHG AL,[SI]
         DEC SI
        MOV [SI],AL
        INC SI

down:   DEC BX
        JNZ UP
        LOOP BACK

        HLT
CODE    ENDS
        END
```

(18) Write an ALP to covert the given BCD number into equivalent Binary.
The BCD number is available at BCD_IN and store the result at BIN_OUT.

(19) Write an Aeembly language Program to find the value of $^{n}C_{r}$
using NEAR procedure