UNIT - III

Date
Page

8086 Microprocessor

- 8086 Microprocessor Features

- It is a 16-bit microprocessor
- It has 16-bit data bus, so it can read data from or write data to memory and ports either 16-bit or 8-bit at a time.
- It has 20 bit address bus and can access up to 2^{20} memory locations (1MB).
- It can support upto 64K I/O ports.
- It provides 14, 16-bit registers.
- It has multiplexed address and data bus $AD_0 - AD_{15}$ and $A_{16} - A_{19}$.
- It requires single phase clock with 33% duty cycle to provide internal timing.
- Prefetches upto 6 instruction bytes from memory and queues them in order to speed up the processing.
- 8086 supports two modes of operation
  (i) Minimum mode
  (ii) Maximum mode.

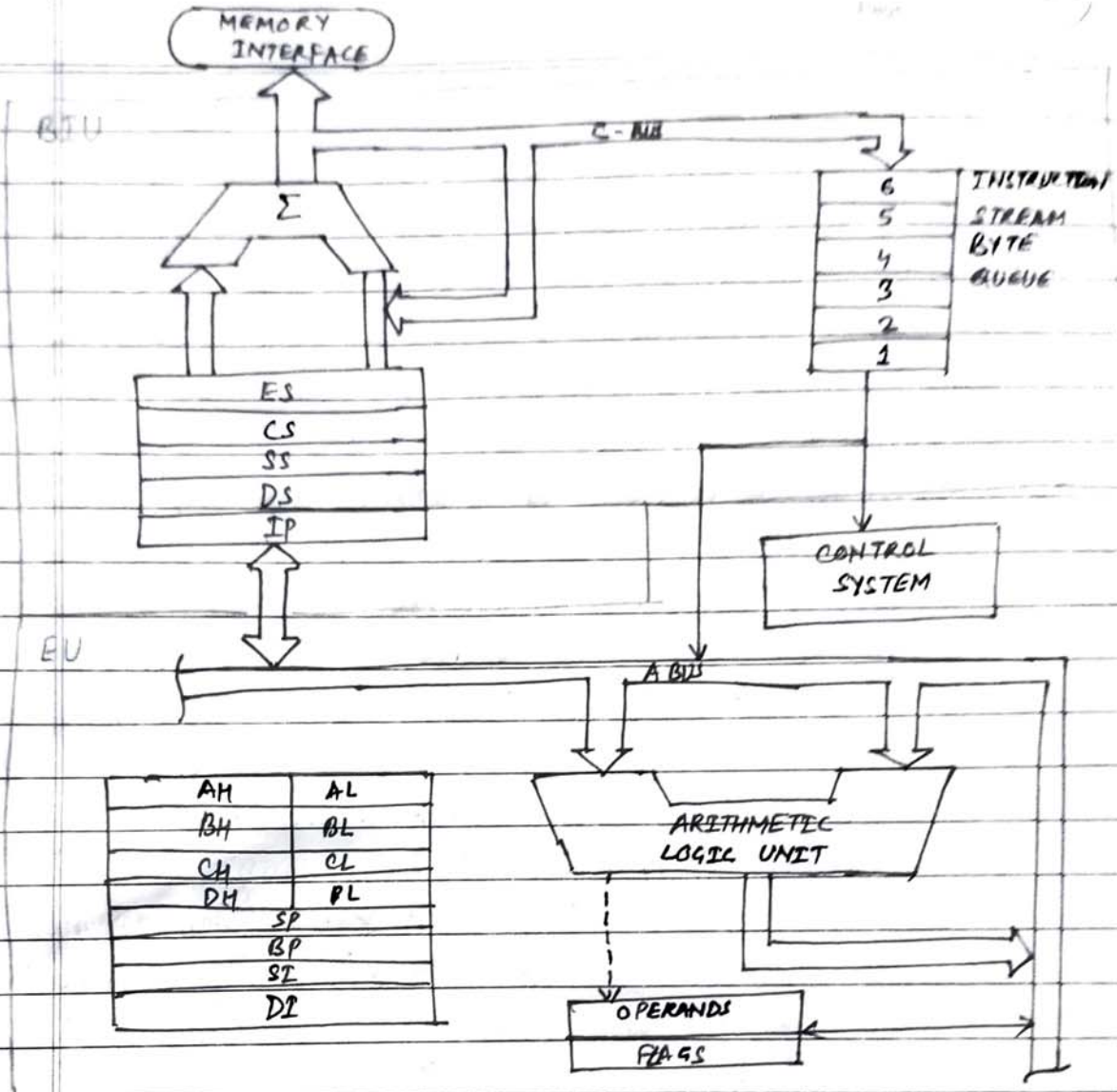
- Architecture of 8086 microprocessor:

The 8086 microprocessor is divided into two independent functional units.

→ Bus Interface Unit (BIU): BIU fetches instructions, reads and writes data, and computes the 20 bit address.

→ Execution Unit (EU): EU decodes and executes the instructions using the 16-bit ALU.

Dividing the work between these two units speed up processing.



Bus Interface Unit (BIU):

→ The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands.
→ The instruction bytes are transferred to the instruction queue.
→ It provides a full 16 bit bidirectional data bus and 20 bit address bus.
→ The BIU is responsible for performing all external bus operations.

→ Specifically it has the following functions:
⇒ Instruction fetch, Instruction queuing, Operand fetch and storage, Address calculation relocation and Bus control.
• The BIU uses a mechanism known as an instruction queue to implement a pipeline architecture.
• This queue permits prefetch of up to six bytes of instruction code. Whenever the queue of the BIU is not full and it has room for at least two more bytes and at the same time EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by prefetching the next sequential instruction.
• These prefetching 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 adding 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.
• EU Accesses the queue from the output end. It reads one instruction byte after the other from the output of the queue. If the queue is full and the EU is not requesting access to the operand in memory.
• If BIU is already in the process of fetching an instruction when EU request it to read or write operand from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read/write cycle.
• The BIU also contains a dedicated adder which is used to generate the 20 bit physical address that is output on the address bus.
This address is formed by adding an appended 16 bit segment address and a 16 bit offset address.

→ BIU has Instruction Pointer. Instruction Pointer is a 16-bit register. This register is always use as effective memory address and is added to Code segment with a displacement of four bits to obtain physical address of opcode.

→ BIU also has segment registers.

• Segment Register: Programmers view memory space as a group of segments defined by application. A segment is a logical unit of memory that may be upto 64K bytes long. Each segment is made up of contiguous memory locations and is an independent addressable unit. Each segment is assigned a base address, which is its starting location in the memory space.

All segments start on 16-bit memory boundaries. Segments may be adjacent, disjoint, partially overlapped or fully overlapped.

The four segment registers are

⇒ Code Segment Register: points to instruction code
⇒ Data Segment Register: points to data memory
⇒ Stack Segment Register: points to Stack memory
⇒ Extra Segment Register: points to data memory.

Execution Unit (EU):

→ Execution unit is responsible for decoding and executing all instructions. During the execution of instruction the EU tests the status and control flags and updates them based on the results of exerting instruction.

→ Execution unit contains control circuitry, which directs internal operations.

→ Execution unit consists of

1. General Data Register 2. Arithmetic Logic Unit 3. Flag Register.

1. General Data Register : The CPU has eight 16-bit general registers. They are divided into two files of four registers each. They are (i) The data register file and (ii) The Pointer and index register file.
- Data Registers in 8086 are AX, BX, CX, DX. These registers are individually accessible as upper and lower halves of data registers as AH, AL, BH, BL, CH, CL, DH, DL.
- Index Register file consists of Stack Pointer (SP), Base Pointer (BP), Source Index (SI) and Destination Index (DI) registers, all are 16-bits.

2. Arithmetic Logic Unit (ALU): ALU is 16-bits wide. It can do the following 16-bit arithmetic and logic operations
- Arithmetic operations are
  → Addition, Subtraction, Multiplication, Division
  Arithmetic operations may be performed on four types of numbers
  → Unsigned binary numbers → Signed binary Numbers
  → Unsigned packed decimal numbers
  → Unsigned unpacked decimal numbers.
- Logical operations are
  → NOT, AND, OR, EXCLUSIVE OR, TEST

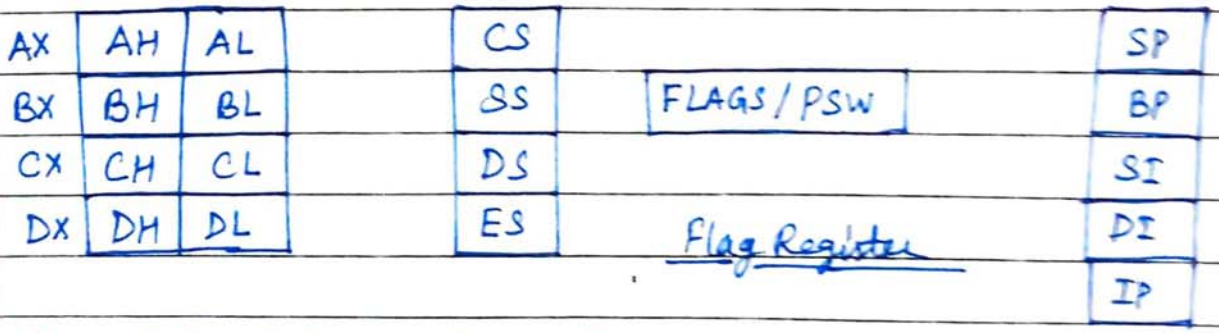
3. Flag Register : The EU has a 16 bit flag register which indicates some conditions affected by the execution of an instruction. The Flag Register in EU contains nine active flags

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AC	X	PF	X	CY
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Out of 9 flags Six are used to indicate some condition produced by an instruction. These condition flags are also called status flags. The conditional flags are:
Carry flag, Parity flag, Auxiliary flag, Zero flag, and Sign flag. The three control flags are Trap flag, Direction flag, Interrupt flag.

REGISTER ORGANIZATION

- 8086 has a powerful set of registers known as general purpose registers and special purpose registers.
- All of them are 16-bit registers.
- General purpose registers:
  - These registers can be used as either 8-bit registers or 16-bit registers.
  - They may be either used for holding data, variables and intermediate results temporarily or for other purposes like a counter or for storing offset address for some particular addressing modes, etc.
- Special purpose registers:
  - These registers are used as segment registers, pointers, index registers or as offset storage registers for particular addressing modes.
- The 8086 registers are classified into the following types:
  - ⇒ General Data Registers
  - ⇒ Segment Registers
  - ⇒ Pointers and Index Registers
  - ⇒ Flag Register



⇒ General Data Registers

The registers Ax, Bx, Cx and Dx are the general data registers.

• Ax Register: Ax is Accumulator register. It consists of two 8-bit registers AL and AH, which can be combined together and used as a 16 bit register Ax.
AL contains lower order byte of the word and AH contains the higher order byte. Accumulator can be used for I/O operations, rotate and string manipulation.

• Bx Register:
* The register is mainly used as a base register.
* It holds the starting base location of a memory region within a data segment.
* It is used as offset storage for forming physical address in case of certain addressing mode.

• Cx Register:
* It is used as default counter or count register in case of string and loop instructions.

• Dx Register:
* Data registers can be used as a port number in I/O operations and implicit operand or destination in case of few instructions. In case of integer 32-bit multiply and divide instruction the Dx register contains high order word of initial or resulting number.

⇒ Segment Registers

The complete 1 Mbyte memory is divided into 16 logical segments. Each segment contains 64 Kbyte of memory. There are four segment registers: register.

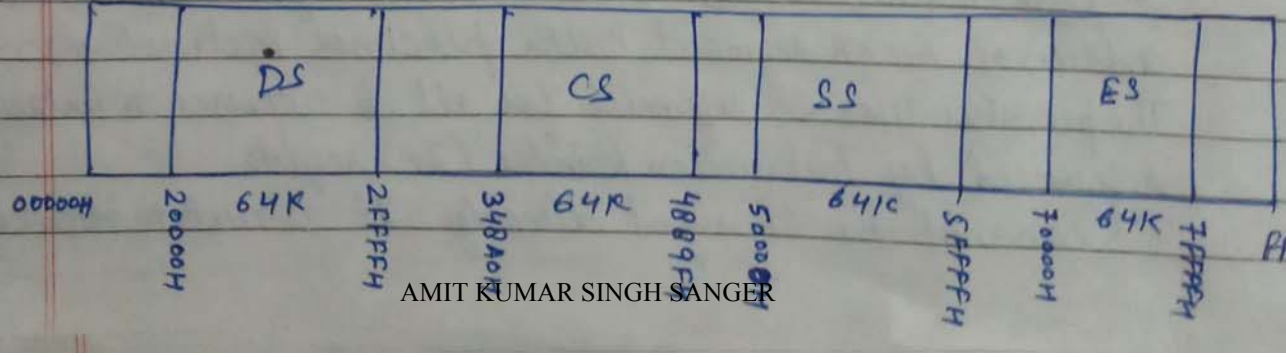
* Code Segment (CS): CS is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by Instruction Pointer (IP) register.
CS can not be changed directly. It is automatically

updated during far jump, far call and far return instructions.
It is used for addressing a memory location in the code segment of memory where the executable program is stored.

* Stack Segment (SS): Register
⇒ It is a 16-bit register containing address of 64 KB segment with program stack.
⇒ By default, processor assumes that all data referenced by the stack Pointer (SP) and Base Pointer (BP) registers is located in the stack segment.
⇒ It is used for addressing stack segment of memory.

* Data Segment (DS): Register
⇒ It is a 16-bit register containing address of 64 KB segments with program data.
⇒ By default, all the data referenced by general register and index register (SI, DI) is located in data segment
⇒ It points to data segment memory.

* Extra Segment (ES) Register
⇒ It is a 16-bit register containing address of 64 KB segment with program data.
⇒ By default DI register references the ES segment in string manipulation instructions.
⇒ It refers to segment which essentially is another data segment of memory. It also contains data.



→ Pointers and Index Registers: The pointers contain within the particular segment. The pointers IP, BP, SP, usually contain offsets within the code, data and stack segments respectively.

- Stack Pointer (SP) is a 16-bit register is used to hold the offset address for stack segment.
- Base Pointer (BP) is a 16-bit register is used to hold the offset address for stack segment.
  (i) BP register is usually used for based, based indexed or register indirect addressing.
  (ii) The difference between SP and BP is that the SP is used internally to store the address in case of interrupt and the CALL instruction.
- Source Index (SI) and Destination Index (DI)

These two 16-bit registers are used to hold the offset address for DS and ES in case of string manipulation instruction.

(i) SI is used for indexed, based indexed and register indirect addressing, as well as a source data addresses in string manipulation instruction.

(ii) DI is used for indexed, based indexed and register indirect addressing, as well as a destination data addresses in string manipulation.

- Instruction Pointer (IP)

→ It is a 16 bit register. It acts as a program counter and is used to hold the offset address for CS.

Flag Register:

- Flag register determines the current state of the processor. They are modified automatically by CPU after arithmatic operations
- Flag register allow to determine the type of result and to determine conditions to transfer control to other parts of the program.
- The 8086 flag register has 9 active flags and they are divided into two categories:
  1. Conditional Flags or Status Flags (06)
  2. Control Flags. (03)
- Conditional or status flags reflect the result of executing an instruction. The programmer cannot set/reset these flags directly.
- Control flags enables or disable certain CPU operations. The programmer can set/reset these bits to control CPU's operation.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AC	X	PF	X	CY

X ⇒ Undefined.

Conditional flags

⇒ Carry Flag (CF): This flag indicates an overflow condition for unsigned integers.
This flag is set to one if the addition of two 16-bit binary numbers produces a carry out of the most significant bit position or if there is a borrow to the MSB after subtraction.
This flag is also affected when other arithmetic and logical instruction are executed.

⇒ Parity Flag (PF): This flag is set, if the result of the operation has an even number of 1s in the lower 8 bits of the result and for odd number of 1s the parity is reset.
This flag can be used to check for data transmission error.

⇒ Auxiliary Carry Flag (AF): This flag is set when there is a carry out of the lower nibble to the higher nibble or a borrow from higher nibble to lower nibble.
The auxiliary carry flag is used for decimal adjust operation. This flag is of significance only for byte operations during which lower order byte of the 16-bit word is used.

⇒ Zero Flag (ZF): This flag is set when the result of an operation is zero. The flag is reset when the result is not zero.

⇒ Sign Flag (SF): This flag is set when an MSB bit of the result is high after an arithmetic operation.
When this flag is set the data is assumed to be negative and when this flag is reset it is assumed to be positive.

⇒ Overflow Flag (OF): This flag is set when an arithmetic overflow occurs. Overflow means that the size of the result exceeded the storage capacity of the destination, and a significant digit has been lost.

Control Flags

The control flags are deliberately set or reset with specific instructions included in the program.

⇒ Trap Flag (TF): It is used for single step control. It allows users to execute one instruction of a program at a time for debugging. When trap flag is set, program can be run in single step mode.

⇒ Interrupt Flag (IF): It is an interrupt enable / disable flag. If it is set, the maskable interrupt of 8086 is enabled and if it is reset, the interrupt is disabled. It can be set by executing instruction sit and can be cleared by executing CLI instruction.

⇒ Direction Flag (DF): It is used in string operation. If it is set, string bytes are accessed from higher memory address to lower memory address. When it is reset, the string bytes are accessed from lower memory address to higher memory address.

Memory Segmentation and Memory addressing

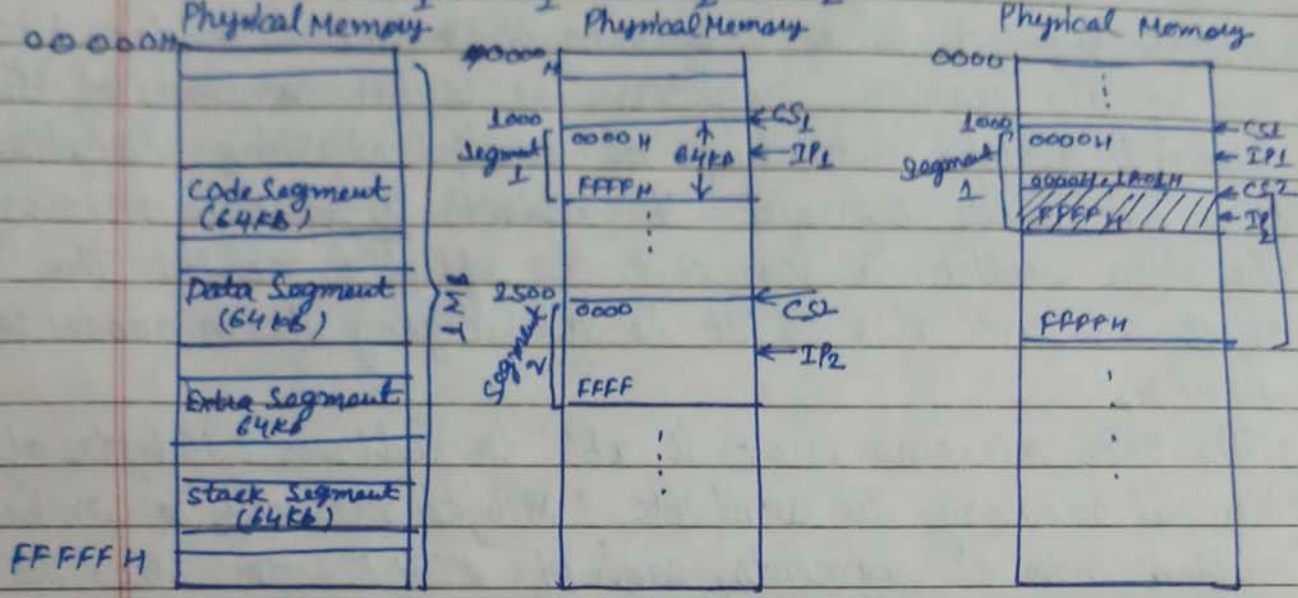
- Memory Segmentation :- The memory in an 8086 based system is organized as segmented memory.
- In this scheme, the complete memory may be divided into a number of logical segments.
- Each segment is 64 K bytes in size and is addressed by one of the segment registers. The 16-bit contents of the segment register actually point to the starting location of a particular segment. To address a specific memory location within a segment an offset is needed. The offset address is also 16 bit long ranging from 0000H to FFFFH.
- The 8086 microprocessor is able to address 1 M bytes of physical memory. The complete 1 M byte memory can be divided into 16 segments, each of 64 K bytes size. The address of segments may be assigned as 0000H to F000H respectively. The offset values are from 0000H to FFFFH so that the physical address ranges from 00000H to FFFFFH.
- There are two types of segments :-
  - → Non-overlapping segments
  - → Overlapping segments.

Non overlapping segments: A segment starts at a particular address and its maximum size can go up to 64 K bytes. If another segment starts after this 64 K byte location of first segment, the two segments are said to be non overlapping segment.

Overlapping segments: A segment starts at a particular address and its maximum size can be 64 K bytes. But, if another segment starts before this 64 K byte locations of first segment, the two segments are said to be overlapping segments. The area of memory from the start of second segment to the possible end of the first segment is called an overlapped segment area.

The locations lying in the overlapped area may be addressed by the same physical address generated from two different sets of segment and offset address.

i.e. $CS_1 + IP_1 = CS_2 + IP_2$



Memory Segmentation | Non overlapping Segments | Overlapping Segments

- The main advantages of the segmented memory scheme are as follows:
  - → Allows the memory capacity to be 1 M bytes although the actual addresses to be are of 16 bit size.
  - → Allows the placing of code, data and stack portions of the same program in different parts of memory, for data and code protection.
  - → Permits a program and/or its data to be put into different areas of memory each time the program is executed.

Physical address generation:

→ The contents of segment register are multiplied by 16 i.e. shifted by 4 position to left by inserting 4 zero bits.
→ Then the offset are added to the shifted contents of segment register to generate physical address.

For example,

1. Segment address = 1005H
   Offset address = 5555H
   Segment address = 1005H = 0001000000000101
   Shifted Segment address = 0001000000000101 0000
   Offset address = + 0101010101010101
   Physical address = 0001010101011010 0101
   = 15A5H

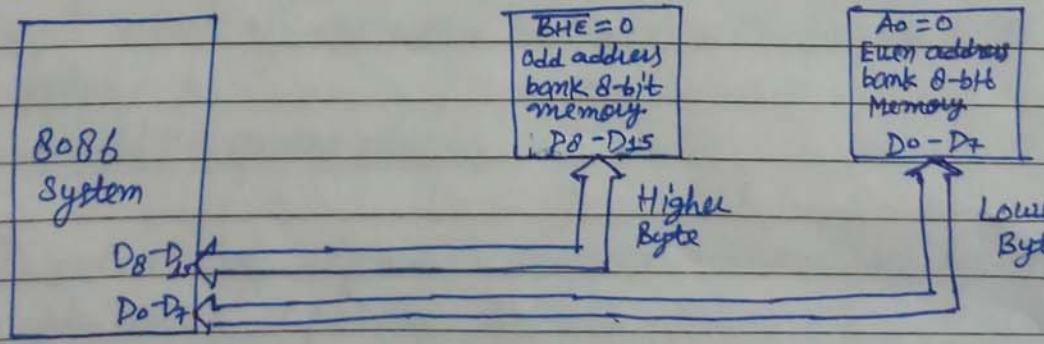
2. CS = 348AH
   IP = 4214H
   Shifted CS = 348A0H
   IP = + 4214H
   Physical address = 38AB4H

Physical Memory Organization

- In 8086 based system, 1 MByte memory is physically organized as odd bank and an even bank, each of 512 Kbytes, addressed in parallel by the processor.
- Byte data with even address is transferred on $D_7 - D_0$, while byte data with odd address is transferred on $D_{15} - D_8$ bus lines.
- The processor provides two enable signals, $\overline{BHE}$ and A_0 for selection of either even or odd or both the banks.
- If the processor fetches a word from memory, there are different possibilities, like:
  1. Both the bytes may be data operands.
  2. Both the bytes may be opcode bits
  3. One of the bytes may be opcode while the other may be data.
- Commercially available memory chips are only byte size.

The 8086 microprocessor is a 16-bit microprocessor, to read/write 16-bit data two successive memory locations are used and lower byte of 16-bit data can be stored in the first memory location while second byte is stored in next location.

A map of 8086 memory system starts at 00000H and ends at FFFFFH. As 8086 is a 16-bit processor, to access 16-bit data two commercially available memory chips are used in parallel.



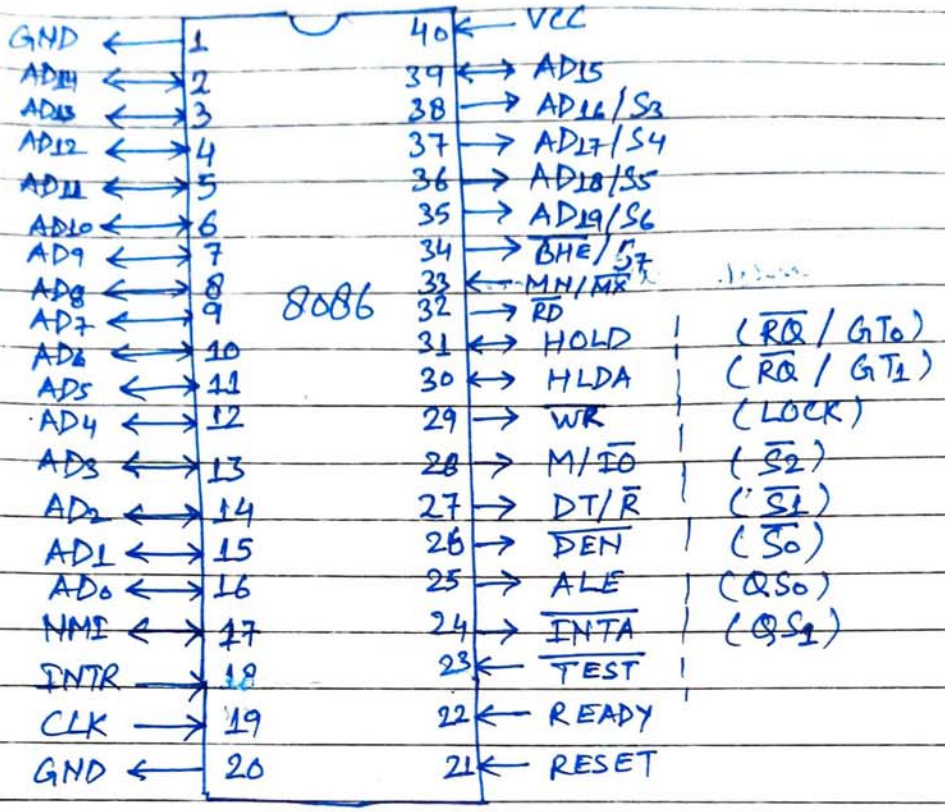
Date
Page

• BIU requires one or two memory cycles, depending upon whether the starting byte is located at an even address or odd address. It is always better to locate the word data at an even address. To read/write a complete word to/from memory, if it is located at an even address, only one read/write cycle is required.
If the word is located at an odd address, the first read/write cycle is required for accessing the lower byte while the second one is required for accessing the upper byte. Thus, two bus cycles are required if a word is located at an odd address.
• The way in which data is read/write is decided by the value of $\overline{BHE}$ and the last address bit, that is the A_0 line. It is done in following way:

$\overline{BHE}$	A_0	Operation performed on memory
0	0	16 bit of data will be read/write into the memory
0	1	8 bit of data will be read/write into odd memory bank.
1	0	8 bit of data will be read/write into even memory bank.
1	1	No operation is performed

Operating Modes

Pin Description of 8086



- $AD_0 - AD_{15}$ (Bidirectional)
  - Address / Data bus
  - Low order address bus; these are multiplexed with data.
  - When AD lines are used to transmit memory address the symbol A is used instead of AD, for example $A_0 - A_{15}$
  - When data are transmitted over AD lines the symbol D is used in place of AD, for example $D_0 - D_7$, $D_8 - D_{15}$ or $D_0 - D_{15}$
- A_{16}/S_3, A_{17}/S_4, A_{18}/S_5, A_{19}/S_6
  - High order address bus. These are multiplexed with status signals.
- $\overline{BHE}/S_7$
  - Bus High Enable / Status
  - It is used to enable data onto the most significant half of data bus $D_8 - D_{15}$. The 8-bit device connected to upper half of data bus use $\overline{BHE}$ signal.
  - It is multiplexed with status signal S_7.

- MN/$\overline{MX}$
  - Minimum / Maximum
  - This pin is used to indicate the operating mode of processor.
- $\overline{RD}$ (Read)
  - It is an active low signal
  - It is an output signal
  - It is used for read operation.
- $\overline{TEST}$
  - $\overline{TEST}$ input is tested by 'WAIT' instruction.
  - 8086 will enter a wait state after execution of WAIT instruction and will resume execution only when the $\overline{TEST}$ is made low by an active hardware.
  - This is used to synchronize an external activity to the processor internal operation.
- READY
  - This is the acknowledgement from the slow device or memory that they have completed the data transfer.
  - The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the 8086.
  - The signal is active high.
- RESET
  - Causes the processor to immediately terminate its present activity.
  - The signal must be active high at least four clock cycl
- INTR (Interrupt Request)
  - This is triggered input. This is sampled during the last clock cycles of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle.
  - This signal is active high and internally synchronized.

- NMI
  - Non Maskable Interrupt line is similar to INTR except that the NMI interrupt does not check to see if the IF flag bit is at logic 1.
  - This interrupt cannot be masked and no acknowledgement is required.
  - It should be reserved for "catastrophic" events such as power failure or memory errors.
- CLK (clock input)
  - Clock input provides the basic timing for processor operation and bus control activity.
  - It is an asymmetric square wave with 33% duty cycle.
- VCC
  - +5V power supply for the operation of the internal circuit
- GND
  - Ground for the internal circuit.
- Minimum mode Signals (Pin 24-31)
  - $DT/\overline{R}$ (Data transmit/Receive)

Output signal from the processor to control the direction of data flow through the data transceivers.

- $\overline{DEN}$ (Data Enable)

Output signal from the processor used as output enable for the transceivers.

- ALE (Address Latch Enable)

Used to demultiplex the address and data lines using external latch.

- $M/\overline{IO}$

Used to differentiate memory access and I/O access. For memory reference instruction it is high. For IN and OUT, it is low.

- $\overline{WR}$

Write Control signal ~~enables~~ the memory that whenever processor writes data to memory or I/O port.

- $\overline{INTA}$ (Interrupt Acknowledgement)
When the interrupt request is accepted by the processor, the output is low on this line.
- HOLD
• Input signal to processor from the bus master as a request to grant the control of the bus.
• Usually used by the DMA controller to get the control of the bus.
- HLDA (Hold Acknowledge)
• Acknowledge signal by the processor to the bus master requesting the control of the bus through HOLD.
• The acknowledge is asserted high, when processor accepts HOLD.

Maximum mode signals (Pin 24-31)
- $\overline{S_2}, \overline{S_1}, \overline{S_0}$: (Status signals)
• Used by the 8086 bus controller to generate bus timing and control signals. These are decoded as shown in table.

Status Signals			Machine Cycle
$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive / Inactive

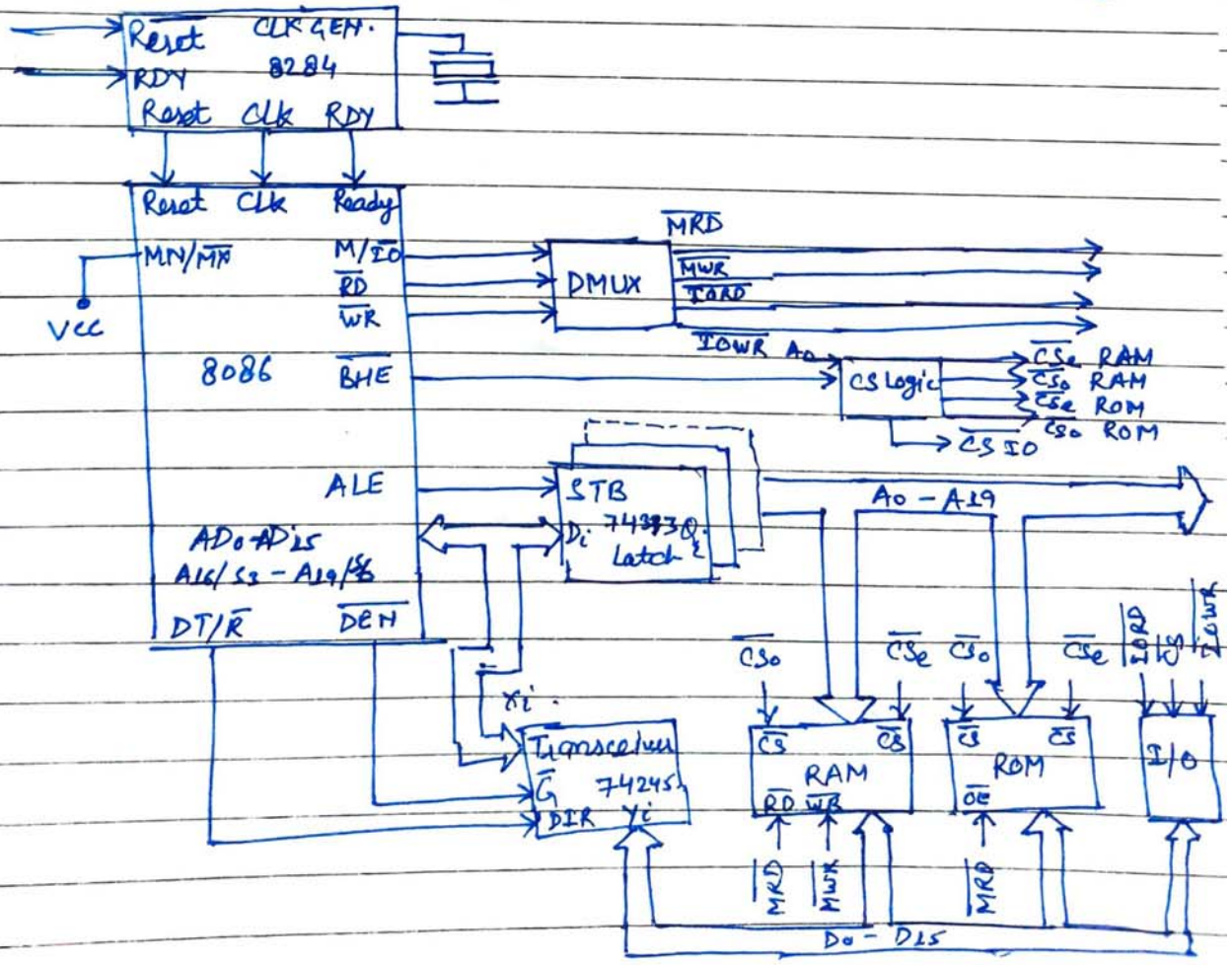
- QS_1, QS_0: (Queue Status)
• The processor provides the status of queue in these lines.
• The queue status can be used by external device to track the internal status of the queue in 8086.
• The output on QS_1 and QS_0 can be interpreted as shown in the table.

Queue Status QS_1	QS_0	Queue Operation
0	0	No operation
0	1	First byte of an opcode from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

- $\overline{RQ}/\overline{GT_0}$, $\overline{RQ}/\overline{GT_1}$: (Bus Request / Bus Grant)
  - These requests are used by other local bus master to force the processor to release the local bus at the end of the processor's current bus cycle.
  - These pin are bidirectional.
  - The request on $\overline{GT_0}$ will have higher priority than $\overline{GT_1}$.
- $\overline{LOCK}$:
  - An output signal activated by the Lock prefix instruction.
  - Remain active until the completion of the instruction prefix by LOCK.
  - The 8086 output low on the $\overline{LOCK}$ pin while execution an instruction prefixed LOCK to prevent other bus masters from gaining control of the system bus.

8086 Minimum Mode System operation

- In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/$\overline{MX}$ pin to logic 1.
- In this mode, there is a single microprocessor. All the control signals are given out by the microprocessor chip itself.
- The remaining components in the system are
  - → Latches
  - → Transceivers
  - → Clock generator
  - → Memory and I/O devices
  - → chip selection logic may be required.
- → The general system organization is shown in below figure:



- The latches are
  - → generally buffered output D flip-flops, like, 74LS373.
  - → used for separating the valid address from the multiplexed address / data lines.
  - → are controlled by ALE signal generated by 8086

  Since, 8086 CPU has 20 address lines and 16 data lines, so it requires three octal address latches and two octal data buffers for the complete address and data separation.
- Transceivers are
  - → the bidirectional buffers and sometimes they are called as data amplifiers.
  - → required to separate the valid data from the time multiplexed address / data lines.
  - → controlled by two signals, $\overline{DEN}$, DT/$\overline{R}$.
    - The $\overline{DEN}$ signal indicates that valid data is available on data bus.
    - DT/$\overline{R}$ indicates the direction of data.
- The system contains memory
  - → for the monitor: EPROMs are used for monitor storage.
  - → users program storage: while RAMs for user program storage.

A system may contain I/O devices for communication with the processor as well as some special purpose I/O devices.

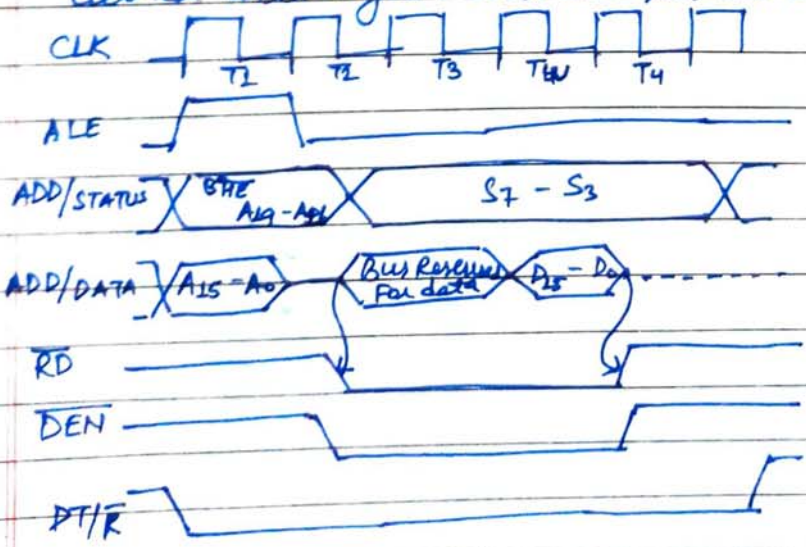
- The clock generator
  - → generates the clock from the crystal oscillator and then shapes it and divides to make it more precise.
  - → also synchronizes some external signals with the system clock.

- The working of the minimum mode configuration system can be better described in terms of the timing diagrams rather than qualitatively describing the operations.

→ The opcode fetch and read are similar.
→ Timing diagram can be categorized in two parts
- timing diagram for read cycle
- timing diagram for write cycle

Read cycle timing diagram for Minimum mode:

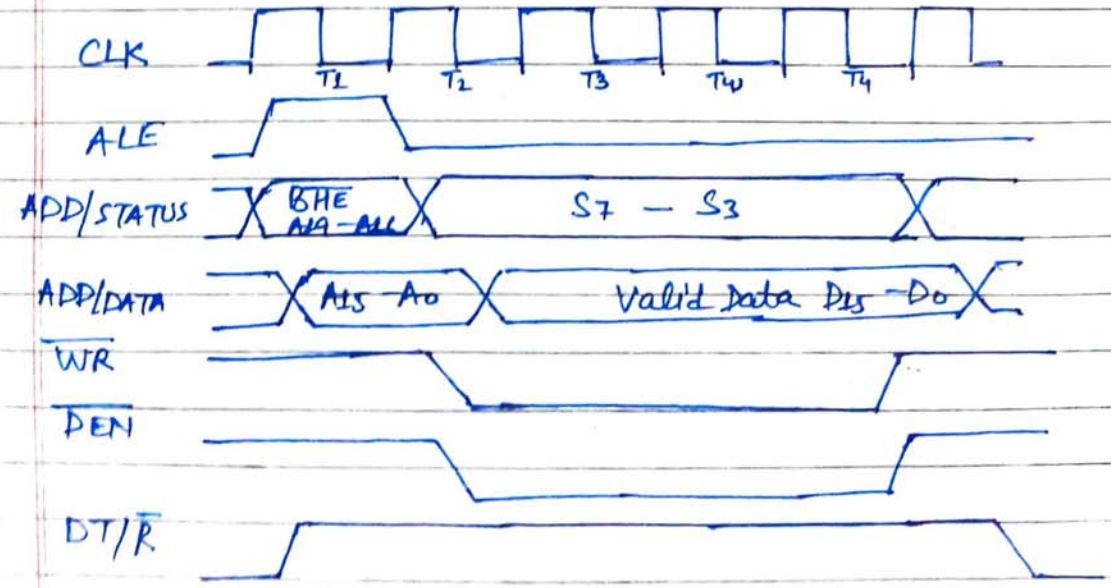
→ The read cycle begins in T_1 with the assertion of the address latch enable (ALE) signal and also M/$\overline{IO}$ signals.
→ During the negative going edge of this signal
* the valid address is latched on the local bus.
* The $\overline{BHE}$ and A_0 signals address low, high or both bytes.
* From T_1 to T_4, the M/$\overline{IO}$ signal indicates a Memory or I/O operation
* At T_2 the address is removed from the local bus and is sent to the output. The bus is then tristated. The read ($\overline{RD}$) control is activated in T_2.
* The read ($\overline{RD}$) signal causes the addressed devices to enable its data bus drivers. After $\overline{RD}$ goes low, the valid data is available on the data bus. The addressed device will drive the READY line high. When the processor returns the read signal to high level the addressed device will again tristate its bus drivers.



- Write cycle timing diagram for Minimum Mode:

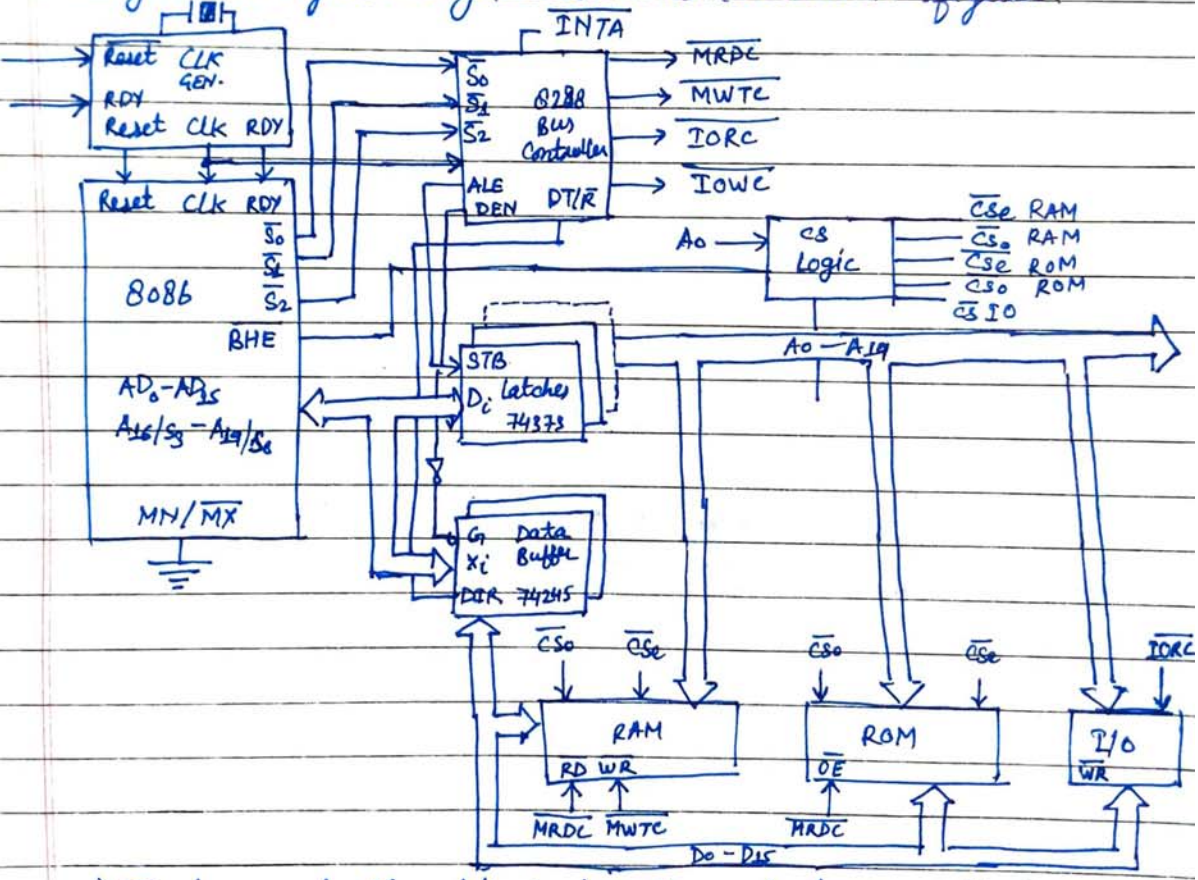
* A write cycle also begins with the assertion of ALE and emission of the address.
* The M/$\overline{IO}$ signal is again asserted to indicate a memory or I/O operation.
* In T_2, after sending the address in T_1, the processor sends the data to be written to the addressed location.
* The data remains on bus until middle of T_4 state.
* The $\overline{WR}$ becomes active at the beginning of T_2.
* The $\overline{BHE}$ and A_0 signals are used to select the proper byte of memory or I/O word to be read or written.
* The M/$\overline{IO}$, $\overline{RD}$, and $\overline{WR}$ signals indicate the type of data transfer as specified in table

M/$\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Transfer type
0	0	1	I/O Read
0	1	0	I/O write
1	0	1	Memory Read
1	1	0	Memory write



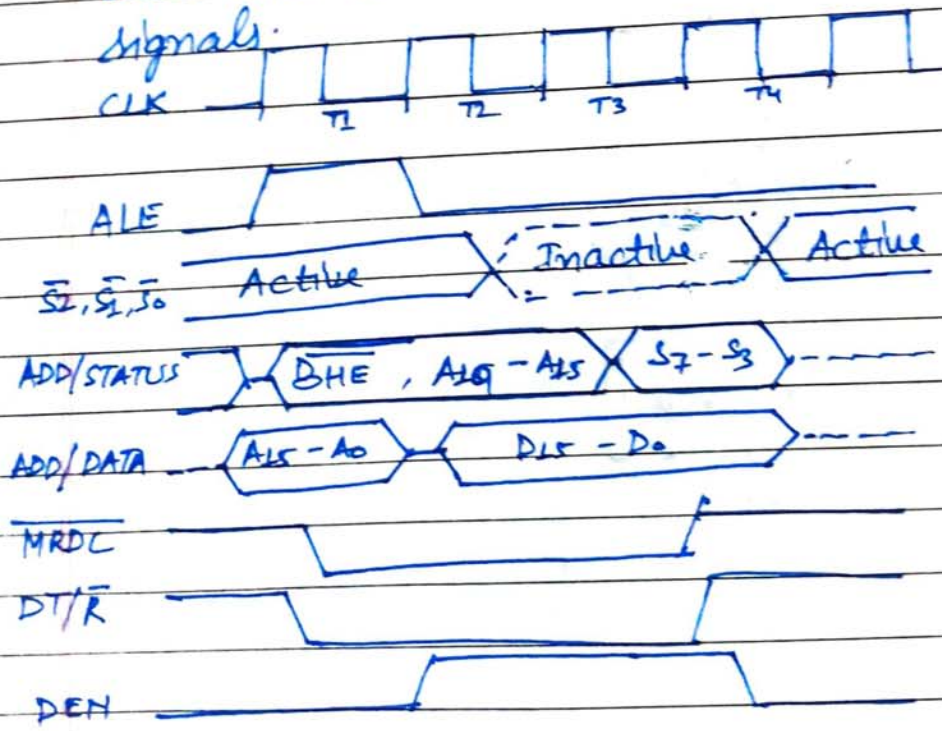
8086 Maximum Mode System Operation

- In the maximum mode, the 8086 is operated by strapping the MN/$\overline{MX}$ pin to ground.
- In this mode, the processor derives the status signals $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$. Another chip called bus controller derives the control signal using this status information.
- In the maximum mode, there may be more than one microprocessor in the system. The other components are same as in the minimum mode system.
- The general system organization is shown in figure:

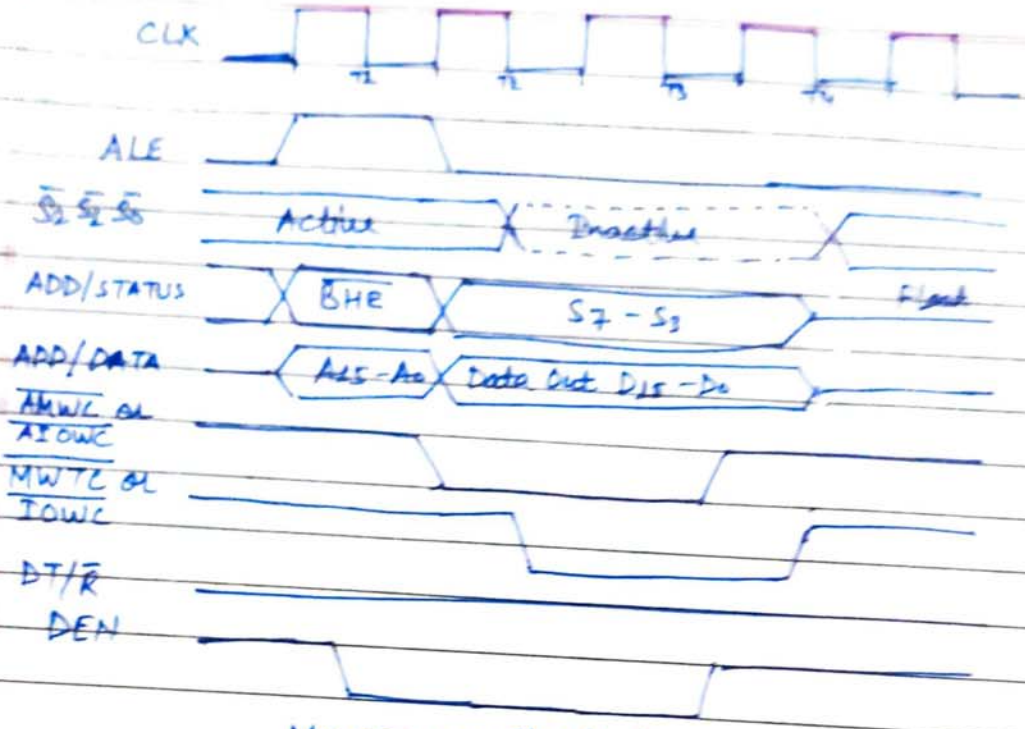


* The bus controller chip derives the control signals like $\overline{RD}$ and $\overline{WR}$, DEN, DT/$\overline{R}$, ALE, etc; using this information made available by the processor on the status lines.
* $\overline{INTA}$ pin is used to issue two interrupt acknowledge pulses to the interrupt controller or to an interrupting device.

- $\overline{IORC}$, $\overline{IOWC}$ are I/O read command and I/O write command signals respectively. These signals enable an IO interface to read or write the data from or to the addressed port.
- The $\overline{MRDC}$, $\overline{MWTC}$ are memory read command and memory write command signals respectively and may be used as memory read and write signals. All commands signals instruct the memory to accept or send data from or to the bus.
- The maximum mode system timing diagrams are also divided into two portions as read and write timing diagram.
- The address/data and address/status timings are similar to the minimum mode. ALE is asserted in T_1, just like minimum mode. The only difference lies in the status signals used and the available control and advanced command signals.



Memory read timing in maximum mode.



Memory write timing diagram in maximum mode

Comparision between Minimum and Maximum mode

S.No	Minimum Mode	Maximum Mode
1.	MN/MX pin is connected to logic 1.	MN/MX pin is connected to logic 0.
2.	CPU generates control signals directly	An external bus controller is needed.
3.	Used for single processor system	Used for multiprocessor system
4.	Least expensive	More expensive
5.	Minimum mode signals (M/IO, ALE, DT/R, DEN, M/IO, WR, HOLD, HLDA)	Maximum mode signals ($\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$, $\overline{LOCK}$, $\overline{RQ}/\overline{GT_0}$, $\overline{RQ}/\overline{GT_1}$)
6.	No provision for multiprocessor configuration.	It supports multiprocessor.

Instruction Sets and Instruction Format

Addressing Modes

- Addressing Mode indicates a way of locating data or operands.
- Depending upon the data types used in instruction and the memory addressing modes, any instruction may belong to one or more addressing modes.
- Addressing modes used in 8086 microprocessor are:

1. Immediate 2. Direct addressing 3. Register addressing
4. Register Indirect addressing 5. Indexed addressing
6. Register Relative addressing 7. Based Indexed addressing
8. Relative Based Indexed addressing

Some of the addressing modes are used for control transfer instructions.

1. Intersegment
   (a.) Intersegment direct
   (b.) Inter segment indirect
2. Intrasegment
   (a.) Intrasegment direct
   (b.) Intrasegment indirect.

1. Immediate addressing mode: In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

eg:- MOV AX, 0005H
MOV BL, 05H

2. Direct addressing mode: In direct addressing mode, a 16-bit memory address (offset) directly specified in the instruction as a part of it.

eg:- MOV AX, [5000H]
Effective address is DS * 10H + offset, i.e.
DS * 10H + 5000H

3 Register addressing mode: In the register addressing mode, the data is stored in a register and it is referred using the particular register. All the register except IP, may be used in this mode.

eg: MOV BX, AX
ADC AL, BL

4. Register Indirect addressing: Sometimes, the address of the memory location which contains data or operand is determined in an indirect way, using the offset registers. This mode of addressing is known as register indirect mode.
In this addressing mode, the effective address of data is in either BX or SI or DI Register. The default segment is either DS or ES.

eg: MOV AX, [BX]
Effective addres = DS * 10H + [BX]

5. Indexed addressing mode: In this addressing mode offset of the operand is stored in one of the indexed registers. DS and ES are default segment registers for index registers SI and DI respectively.

eg: MOV AX, [SI]
Effective addres = DS * 10H + [SI]

6. Register Relative Addressing mode: In this addressing mode the data is available at an effective address formed by adding an 8 bit or 16-bit displacement with the content of any register BX, BP, SI, and DI in the default segment.

eg: MOV AX, 50H[BX]
Effective addres = DS * 10H + 50H + [BX]

7. Based Indexed addressing mode: The effective address of data is formed in this addressing mode by adding content of a base register to the content of an index register. The default segment registers may be ES or DS.

eg: MOV AX, [BX][SI]
Effective Address = DS * 10H + [BX] + [SI]

8. Relative Based Indexed addressing mode: The effective address is formed by adding an 8 bit or 16 bit displacement with the sum of contents of any of the base register (BX or BP) and any of the index registers, in a default segment.

eg: MOV AX, 50H[BX][SI]

Effective address = DS * 10H + [BX] + [SI] + 50H.

- For Control transfer instruction

- If location to which the control is to be transferred lies in a different segment other than the current one, the mode is called intersegment mode.
- If destination location lies in the same segment, the mode is called intrasegment mode.

1(a) Intersegment direct: In this mode the address to which the control is to be transferred in a different segment. This addressing mode provides a mean of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

eg: JMP 5000H:2000H.

(b) Intersegment indirect: In this mode, the address to which the control is to be transferred lies in a different segment and is passed to the instruction indirectly i.e. contents of memory block containing four bytes, i.e. IP(LSB), IP(MSB), CS(LSB), CS(MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode

eg: JMP [2000H]

2.(a) Intrasegment direct: In this mode the address to which the control is to be transferred lies in the same segment in which the control transfers instruction lies and appears directly in the instruction as an immediate displacement value. In this

addressing mode, the displacement is computed relative to the content of instruction pointer.

The effective address to which the control will be transferred is given by the sum of 8 or 16 bit displacement and current content of IP. In case of jump instruction, if the signed displacement (d) is of 8-bit (i.e $-128 < d < +127$), it as short jump and if it is of 16 bits i.e. $(-32768 < d < +32767)$, it is termed as long jump.

eg:- JMP SHORT LABEL

(b.) Intrasegment indirect: In this mode, the displacement to which the control is to be transfered is in the same segment in which the control transfer instruction lies, but it is passed to the instruction directly. Here, the branch address is found as the content of register or a memory location.

This addressing mode may be used in unconditional branch instruction.

eg:- JMP [BX]

Machine language Instruction Format 8086

- A machine language instruction format has one or more number of fields associated with it
- The first field is called as operation code field or opcode field
- The other fields known as operand fields
- The CPU executes the instruction using the information which resides in these fields
- There are six general formats of instructions in 8086 instruction set. The length of an instruction may vary from one byte to six bytes. The instruction formats are described as follows:

1. One byte instructions: This format is only one byte long and may have the implied data or register operand. The least significant 3 bits of opcode are used for specifying the register operand, if any. Otherwise all 8 bits form an opcode and the operands are implied.

2. Register to Register: This is 2 byte long. The first byte of the code specifies the operation code and width of the operand specified by w bit. The second byte of the code shows the register operands and R/M field.

D_7	D_2	D_0
OP CODE	W	

D_7 D_6	D_5 D_4 D_3	D_2 D_1 D_0
1 1	REG	R/M

REG ⇒ Register as one of operands

R/M ⇒ Another register or memory location i.e the other operand

3. Register to/from Memory with no Displacement: This format is also 2 bytes long and similar to the register to register format except for the MOD field. The MOD field shows the mode of addressing.

D_7	D_2	D_0
OP CODE	W	

D_7 D_6	D_5 D_4 D_3	D_2 D_1 D_0
MOD	REG	R/M

- Table is given for deciding MOD, REG, R/M, and W fields.

Operand / MOD / R/M	No Displacement 00	Memory operand Displacement 8-bit 01	Displacement 16-bit 10	Register operand 11 W=0	W=1
000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16	AL	AX
001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16	CL	CX
010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16	DL	DX
011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16	BL	BX
100	(SI)	(SI) + D8	(SI) + D16	AH	SP
101	(DI)	(DI) + D8	(DI) + D16	CH	BP
110	D16	(BP) + D8	(BP) + D16	DH	SI
111	(BX)	(BX) + D8	(BX) + D16	BH	DI

4. Register to / from Memory with Displacement: This type of instruction format contains one or two additional bytes for displacement along with 2-byte format of the register to / from memory without displacement.

D7 D0	D7 D6	D5 D4 D3	D2 D1 D0	D7 D0	D7 D0
OPCODE	MOD	REG	R/M	Lower Byte of Displacement	Higher Byte of Displacement

5. Immediate Operand to Register: In this format, the first byte as well as the 3-bits from the second byte which are used for REG field in case of register to register format are used for opcode. It also contains one or two bytes of immediate data.

D7 D0	D7 D6	D5 D4 D3	D2 D1 D0	D7 D0	D7 D0
OPCODE	11	OPCODE	R/M	Lower Byte Data	Higher Byte Data

6. Immediate operand to Memory with 16 bit Displacement: This type of instruction format requires 5 or 6 bytes for coding. The first 2 bytes contain the information regarding OPCODE, MOD, and R/M fields. The 4 bytes contain 2 bytes of displacement and 2 bytes of data.

D7 D0	D7 D6	D5 D4	D3 D2 D1 D0	D7 D0	D7 D0
OPCODE	MOD	OPCODE	R/M	Lower Byte of Disp.	Higher Byte of Disp.

Lower Byte of DATA	Higher Byte of DATA

Instruction Set of 8086

The 8086 instructions are categorised in to the following main types.

(i) Data Transfer/copy instructions: These type of instructions are used to transfer data from source operand to destination operand. All the store, load, move, exchange, input and output instructions belong to this category.

- Instructions to transfer a word

MOV - Used to copy the byte or word from the provided source to destination.
PUSH - Used to put a word at the top of the stack.
POP - Used to get a word from top of the stack to the provided location.
XCHG - Used to exchange the data from two locations.
XLAT - Used to translate a byte in AL using a lookup table in memory.

- Instructions for input and output port transfer

IN - Used to read a byte or word from the provided port to the accumulator.
OUT - Used to write a byte or word from the accumulator to the provided port.

- Instructions to transfer the address

LEA - Used to load the address of operand into the provided register.
LDS - Used to load DS register and other provided register from the memory.
LES - Used to load ES register and other provided register from the memory.

- Instructions to transfer flag registers.

LAHF - Used to load AH with the low byte of the flag register.
SAHF - Used to store AH to low byte of the flag register.
PUSHF - Used to copy the flag register at the top of the stack.
POPF - Used to copy a word at the top of the stack to the flag register.

(ii) Arithmetic Instructions: These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.

- Instruction to perform addition

ADD - Used to add the provided byte to byte/word to word.
ADC - Used to add with carry.
INC - Used to increment the provided byte/ word by 1.
AAA - Used to adjust ASCII after addition.
DAA - Used to adjust the decimal after addition / subtraction.

- Instruction to perform subtraction

SUB - Used to subtract the byte from byte/word from word.
SBB - Used to perform subtraction with borrow.
DEC - Used to decrement the provided byte/word by 1.
CMP - Used to compare two provided byte/word.
AAS - Used to adjust ASCII codes after subtraction.
DAS - Used to adjust decimal after subtraction.

- Instruction to perform multiplication

MUL - Used to multiply unsigned byte by byte/word by word
IMUL - Used to multiply signed byte by byte/word by word.
AAM - Used to adjust ASCII codes after multiplication.

- Instruction to perform division

DIV - Used to divide the unsigned word by byte or unsigned double word by word.
IDIV - Used to divide the signed word by byte or signed double word by word.
AAD - Used to adjust ASCII codes after division.
CBW - Used to fill upper byte of the word with the copies of signed bit of the lower byte.
CWD - Used to fill the upper word of the double word with the sign bit of the lower word.

(iii) Logic Instruction: These instructions are used to perform logic operations like AND, OR, NOT, etc.

AND - Used for performing logical AND operation of two operands and places result in the specific register.

OR - Performs logical OR operation of two operands and places result in specified destination.

NOT - Used to invert each bit of a byte or word.

XOR - Used to perform Exclusive OR operation over each bit in a byte/word with the corresponding bit in another byte/word.

• Instruction to perform shift operations

SHL/SAL - Used to shift bits of a byte/word towards left and put zero in LSBs.

SHR - Used to shift bits of a byte/word towards the right and put zero in MSBs.

SAR - Used to shift bits of a byte/word towards right and copy old MSB into the new MSB.

• Instruction to perform rotate operations

ROR - Used to rotate bits of byte/words towards the right, i.e. LSB to MSB and to Carry Flag (CF).

ROL - Used to rotate bits of bytes/words towards the left, i.e. MSB to LSB and to Carry Flag (CF).

RCR - Used to rotate bits of byte/word towards the right i.e. LSB to (CF) and (CF) to MSB.

RCL - Used to rotate bits of byte/word towards the left i.e. MSB to (CF) and (CF) to LSB.

(iv) Branch Instructions: These instructions transfer control of execution to the specified address. All call, jump, return, interrupt instructions belong to this class.

CALL - Used to call a procedure and save their return address to stack.
RET - Used to return from the procedure to main program.
JMP - Used to jump to the provided address to proceed to the next instruction.

• Instructions to transfer the control conditionally:
JA/JNBE - Used to jump if above / not below / equal instruction satisfies.
JAE/JNB - Used to jump if above / not below instruction satisfies.
JBE/JNA - Used to jump if below / equal / not above instruction satisfies.
JC - Used to jump if carry.
JE/JZ - Used to jump if equal / zero.
JG/JNLE - Used to jump if greater / not less or equal.
JGE/JNL - Used to jump if greater / not less or equal.
JL/JNGE - Used to jump if less / not greater than / equal
JLE/JNG - Used to jump if less than / equal / if not greater than
JNC - Used to jump if no carry.
JNE/JNZ - Used to jump if not equal / zero
JNO - Use to jump if no overflow.
JNP/JPO - Used to jump if not parity / parity odd
JNS - Used to jump if not sign.
JO - Used to jump if overflow.
JP/JPE - Used to jump if parity / parity even.
JS - Used to jump if sign.

• Interrupt Instructions
INT - Used to interrupt the program during execution and calling service specified.
INTO - Used to interrupt the program during execution if overflow, OF = 1.
IRET - Used to return from interrupt service to the main program.

(V.) Loop Instructions: If these instructions have REP prefix with CX used as count register, they can be used to implement unconditional and conditional loops. The LOOP, LOOPNZ, LOOPZ instruction belong to this category.

LOOP - Used to loop a group of instructions until the condition satisfies i.e. CX=0

LOOPE / LOOPZ - Loop while equal or zero, i.e. ZF=1 and CX=0

LOOPNE / LOOPNZ - Loop while not equal or not zero, i.e. ZF=0 and CX=0

JCXZ - Jump if CX=0

(vi) Machine Control Instruction: These instructions control machine status. NOP, HLT, WAIT and LOCK instruction belongs to this group.

NOP - Used to perform No operation.

HLT - Used to Halt processing. It stops program execution.

WAIT - When WAIT instruction is executed, the processor enters an idle state in which the processor does no processing.

LOCK - It is a prefix instruction. It makes the LOCK pin low till the execution of next instruction.

(vii) Flag manipulation instruction: All the instructions which directly affect the flag register come under this group of instructions.

Instructions like CLD, STD, STI, etc. belongs to this group.

CLC - Used to clear/reset carry flag CF=0.

CLD - Used to clear/reset the direction flag DF to 0.

CLI - Used to clear the interrupt enable flag to 0 i.e. disable INTR input.

CMC - Used to put complement at the state of carry flag CF.

STC - Used to set carry flag CF to 1.

STD - Used to set direction flag to 1.

STI - Used to set the interrupt enable flag to 1 i.e. Enable INTR input.

(viii) String Instructions: These instructions involve various string manipulation operations like load, move, scan, compare, store, etc. These instructions are only to be operated upon the string.

REP - Used to repeat the given instruction till CX ≠ 0.

REPE/REPZ - Used to repeat the given instruction till CX ≠ 0 and ZF=1.

REPNE/REPNZ - Used to repeat the given instruction till cx ≠ 0 and ZF = 0.

MOVS/MOVSB/MOVSW - Moves 8-bit or 16-bit data from the memory location addressed by SI to the memory location addressed by DI register

CMPS/CMPSB/CMPSW - Compares the contents of memory location addressed by DI with the contents of memory location addressed by SI register.

SCAS/SCASB/SCASW - Used to scan a string and compare its byte with a byte in AL or string word in AX.

LODS/LODSB/LODSW - Used to load AX/AL register by the contents of string pointed by SI register.

STOS/STOSB/STOSW - Used to store AX/AL contents to a location in the string pointed by DI register.

Types of Instructions

The 8086 instructions are categorised into following main types:

1. Data transfer / copy instructions

The following instructions come under data transfer/copy instructions:

- MOV
- PUSH
- POP
- IN
- OUT
- PUSHF
- POPF
- LEA
- LDS/LES
- XLAT
- XCHG
- LAHF
- SAHF

MOV: This instruction transfer data from one register/memory location to another register/memory location

Source ⇒ Any one of segment / general purpose / special purpose register/ memory location

Destination ⇒ Another register/ memory location

Syntax (1.) MOV mem/reg1, reg2/mem

eg: MOV BX, 0201H ; | MOV [SI], [BX] not valid

MOV AL, BL | No memory to memory transfer.

(2.) MOV mem, DATA

eg: MOV[BX], 02H

(3.) MOV reg, DATA

eg: MOV AL, 11H MOV CX, 1234H

(4.) MOV A, mem ⇒ [A] ← [mem]

eg: MOV AL, [SI] MOV AX, [DI]

(5.) MOV mem, A ⇒ [mem] ← A

eg: MOV [SI], AL MOV [SI], AX

(6.) MOV segmentRegister, mem/Reg

eg: MOV SS, [SI]

(7.) MOV mem/reg, SegmentRegister

eg: MOV DX, SS

Note: In case of immediate mode data, a segment register can not be destination register.

eg: Load DS with 5000H

MOV DS, 5000H (Not permitted) | MOV AX, 5000H

MOV DS, AX

• PUSH: This instruction pushes the contents of specified register/ memory location on stack. The stack pointer is decremented by 2 after each execution of the instruction.

Syntax: PUSH reg

eg: → PUSH AX → PUSH [5000H] → PUSH DS

• POP: This instruction when executed, loads the specified register/ memory with the contents of the memory location on which address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2.

Syntax: → POP mem | → POP reg

eg: POP [5000H] | POP AX

• XCHG: This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them may be memory location.

Syntax:	XCHG AX, reg16	XCHG mem, reg	XCHG reg, reg
eg:	XCHG AX, DX	XCHG [BX], DX	XCHG AL, BL XCHG CX, DX

Other examples are: XCHG [5000H], AX

XCHG BX

• IN: This instruction is used for reading an input port. The address of the input port may be specified in the instruction directly or indirectly.

AL and AX are allowed destination for 8-bit and 16-bit input operations.

DX is the only implicit register which can carry port address.

Syntax: → IN AL, DX → IN AX, DX → IN AL, PORT

→ IN AX, PORT → IN AX (DX is implicit register to have port address)

eg: IN AL, 03C0H

• OUT: This instruction is used for writing to an output port. The address of the output port may be specified in the instruction directly or implicitly in DX.

The content of AL/AX are transferred to addressed port after execution of this instruction.

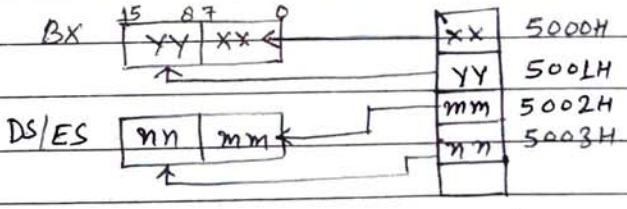
eg: → OUT DX, AL → OUT DX, AX → OUT PORT, AL
→ OUT PORT, AX → OUT AX (Sends Data to a port specified in DX)

• PUSHF: The push flag instruction pushes the flag register on to the stack; first upper byte and then lower byte is pushed on to it. The SP is decremented by 2 for each push instruction.

• POPF: The pop flag instruction loads the flag register completely from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by 2 for each pop operation.

• LEA: The load effective address instruction loads the effective address formed by destination operand into the specified source register.
eg: → LEA BX, ADR → LEA BX, [DI]

• LDS/LES: This instruction loads the DS or ES register and specified destination register in the instruction with the contents of memory location specified in the instruction.
eg: LDS BX, 5000H / LES BX, 5000H



• XLAT: Execution of XLAT replaces the contents of AL by the contents of the accessed lookup table location.
i.e. ((DS)*10H + (BX) + (AL)) ⟶ (AL)

• LAHF: This instruction loads AH register with the lower byte of the flag register. This instruction is used to observe the status of all condition code flags at a time.

• SAHF: This instruction sets or resets the condition code flags in the lower byte of the flag register depending upon the corresponding bit position in AH.

2 Arithmetic Instruction: These instructions usually perform the arithmetic operations like addition, subtraction, multiplication and division along with respective ASCII and decimal adjust instructions

The following instructions are used in this type of instructions:

• ADD • ADC • SUB • SBB • MUL • IMUL • DIV • IDIV • CMP • NEGATE • INC • DEC • DAA • DAS • AAA • AAS • AAM • AAD • CBW • CWD

(a) ADD: → This instruction adds an immediate data or contents of a memory location specified in the instruction or a register to content of another register or memory operands.

→ Both source and destination operands can not be memory operands

→ Contents of segment register can not be added using this instruction.

→ All condition flags are affected depending on the result.

Syntax: ⇒ ADD mem/reg1, mem/reg2

eg: • ADD BL, [SI] • ADD AX, BX

⇒ ADD mem, DATA eg: ADD [SI], 0715H

⇒ ADD reg, DATA eg: ADD CL, 05H

⇒ ADD A, DATA eg: ADD AX, 2020H

(b) ADC: → This instruction performs the same operation as ADD instruction; But adds the carry flag to the result. All conditional flags are affected depending on the result.

Syntax: ⇒ ADC mem/reg1, mem/reg2 eg: ADC BL, [SI]

eg: ADC AX, BX

⇒ ADC mem, DATA eg: ADC [SI], 05H

⇒ ADC reg, DATA eg: ADC AX, 1717H

(C) SUB: → Substracts source operand from destination operand and result left in destination operand.

→ Source and destination both can not be memory operand

→ Destination can not be immediate data.

→ All flags are affected by this instruction.

Syntax: ⇒ SUB mem/reg1, mem/reg2
eg: SUB AX, BX

⇒ SUB mem, DATA eg: SUB [SI], 0721H

⇒ SUB A, DATA eg: SUB AX, 1212H

(d.) SBB: Subtracts source operand and the borrow flag from the destination operand.
The result is stored in destination operand. All the flags are affected by this instruction.

Syntax: ⇒ SBB mem/reg1, mem/reg2
eg: SBB BL, [SI]

⇒ SBB mem, DATA eg: SBB [SI], 7234H

⇒ SBB reg, DATA eg: SBB CL, 05H

⇒ SBB A, DATA eg: SBB AX, 1211H

(e.) INC: • This instruction increments the contents of the specified register or memory location by 1. All the condition flags are affected except the carry flag.

Syntax: ⇒ INC reg16 eg: INC BX

⇒ INC mem/reg8 eg: INC BL
INC SI

Note: Segment register can not be incremented.

(f.) DEC: • This instruction subtract 1 from the contents of the specified register or memory location.
• All conditional flags except carry flag are affected depending upon the result.

Syntax: ⇒ DEC reg16 eg: DEC BX

⇒ DEC mem/reg8 eg: • DEC BL

Note: Segment register can not be decremented.

(g.) MUL: This instruction multiplies unsigned byte or word by the contents of AL. The unsigned byte/word may be in any one of the general purpose register or memory locations. The most significant word of result is stored in DX, while least significant word of result is stored in AX. All the flags are modified depending upon the result.

The result of execution of MUL
(AX) ← (AL) x (8 bit operand)
(DX, AX) ← (AX) x (16 bit operand)
Syntax: ⇒ MUL mem/reg eg: MUL BL ; AX ← AL * BL
MUL BX ; DX, AX ← (AX) * (BX)

(h) IMUL: This instruction multiplies a signed byte in source operand by a signed byte in AL or signed word in source operand by signed word in AX.
Source can be a general register, memory operand, index/base register.
· In case of 32 bit result higher order word is stored in DX and lower order word is stored in AX.
CF and OF flags are affected and SF, ZF, AF, PF are undefined.
Syntax: IMUL mem/reg
eg: → IMUL BL → IMUL BX

(i) DIV: This instruction performs unsigned division. It divides an unsigned word or double word by 16 bit or 8-bit operand.
· For 16-bit operation; Dividend → AX, division may be specified using any one of addressing modes.
Quotient → AL, Remainder → AH.
if result is too big to fit in AL, type 0 (Divide by zero) interrupt is generated.
· For 32-bit operation; Higher word → DX, Lower word → AX
Quotient → AX, Remainder → DX.
This instruction does not affect any flag.
Syntax: DIV mem/reg
eg: DIV BL ⇒ (AX)/(BL)
DIV BX ⇒ (DX)(AX)/(BX)

(j) IDIV: This instruction performs the same operation as DIV, but it with signed operands, the results are stored similarly as in case of DIV. The result will also be signed number.
eg: → IDIV BL → IDIV BX

(k.) AAA: The ASCII Adjust after addition instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL.
→ AAA converts the resulting contents of AL to unpack decimal digits.
→ If AL contain valid BCD number and AF is zero, AAA sets 4 higher bits of AL to 0. The AH is cleared before addition.
→ If lower nibble of AL is between (0 to 9) and AF is set, add 06 to AL. The upper 4 bits are cleared AH is incremented by 1.
→ If lower nibble of AL is greater than 9 then add 06 to AL AH is incremented by 1, the AF and CF are set to 1 and higher 4 bit of AL is cleared to 0.

eg: 1. AL [57] Before to AAA
AL [07] After AAA

2. AL [5A] } Befor AAA
AH [00] }

Lower nibble of AL is >9 so, 1010 + 0110 = 10000
= 10H

AX = [01] [00] After AAA

(l.) AAS: It correct the results in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format.
→ If lower 4 bits of AL register are greater than 9 or if the AF is one, the AL is decremented by 06 and AH is decremented by 1.
CF = 1, AF = 1.
Otherwise, the CF and AF are set to 0 the result need no correction. As a result, upper nibble of AL is 0.
The procedure is similar to the AAA instruction.

(m.) AAM: This instruction, after execution, converts the product available in AL into unpacked BCD format. This follows a multiplication instruction. The lower byte of result remains in AL and higher byte of result remains in AH.

eg:- Product available in AL = 5D H. After AAM BCD result in AX,

DH > 9, ⇒ 1101 + 1100 = 13H

AL = 13H , AH = 5H = 6

AH = 06 AL = 03

(n.) AAD: The AAD instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL.

- Before Division first clear the leftmost nibble in each byte
- Operations done by AAD instruction
  - → AAD multiplies the AH by 10 (0AH)
  - → Then adds the product to AL and clears to AH.
  - → After AAD division is performed.

eg:- AX = 05H 08H

AAD result in AL 00 3AH

(o.) DAA: This instruction is used to convert the result of addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL.

→ If lower nibble is greater than 9 after addition or AF = 1, it will add 06 to lower nibble in AL.

After adding 06 to the lower nibble of AL if upper nibble of AL is greater than 9 or if carry flag is set, DAA instruction adds 60 in AL.

eg:- AL = 73 CL = 29

ADD AL, CL ; AL ← AL + CL = 9C

DAA ; AL = 02 , CF = 1

(p.) DAS: This instruction converts the result of subtraction of two packed BCD numbers to a valid BCD. The subtraction has to be in AL

→ If lower nibble of AL is greater than 9, subtract 06 from lower nibble of AL.

→ If subtraction results set the carry flag or if upper nibble is greater than 9 it subtracts 60H from AL.

Flags modified are AF, CF, PF and ZF.

eg: AL = 38 CH = 61
SUB AL, CH ; AL ← D7H, CF = 1
DAS ; AL ← 77H, CF = 1

(9.) NEG: → It subtracts the contents of destination from zero.
→ It forms 2's complement of specified destination
→ If OF = 1, operation can not be performed.
→ all condition code flags are affected.

(10.) CBW: → Converts a signed byte to a signed word.
It copies sign bit of a byte to all bits in higher byte of the result word.

eg: AL = 08H
CBW ;

AH	AL
00000000	00001000

0 0 08H

(11.) CWD: It copies sign bit of AX to all bits of DX register. This operation is done before signed division.

eg: AX = 102FH

CWD

DX	AX
0000 0000 0000 0000	0001 0000 0010 1111

DX: 0000H
AX: 102FH

3. Logic Instructions: These instructions are used for carrying out the bit by bit shift, rotate or basic logical operations. All the conditional code flags are affected depending upon the result.

- AND
- OR
- NOT
- XOR
- SHL/SAL
- SHR
- SAR
- ROR
- ROL
- RCR
- RCL

(1.) AND: This instruction bit by bit ANDs the source operand that may be an immediate, a register, or a memory location to the destination operand that may be in register or memory location.
→ Both operands can not be memory location or immediate operand.

Syntax: (i) AND mem/Reg1, mem/Reg2
eg:- AND BL, CL

(ii) AND mem, data eg:- AND START, 05H

(iii) AND reg, data eg:- AND AL, F0H

(iv) AND A, data eg:- AND AX, 1022H

(ii) OR: This instruction carries out OR operation in the same way as in case of AND operation.

Syntax: (i) OR mem/Reg1, mem/Reg2 eg:- OR BL, CH

(ii) OR mem, DATA eg:- OR START, 05H

(iii) OR reg, DATA eg:- OR AL, 05H

(iv) OR A, DATA eg:- OR AX, 1234H

(iii) NOT: This instruction complements the contents of an operand register or a memory location bit by bit.

Syntax: (a) NOT reg eg:- NOT AX

(b) NOT mem eg:- NOT [SI]

(iv) XOR: The XOR operation is again carried out in a similar way to the AND and OR operation. The constraints on high output, when two input bits are dissimilar, otherwise, the output is zero.

Syntax: (i) XOR mem/Reg1, mem/Reg2
eg:- XOR BL, CL

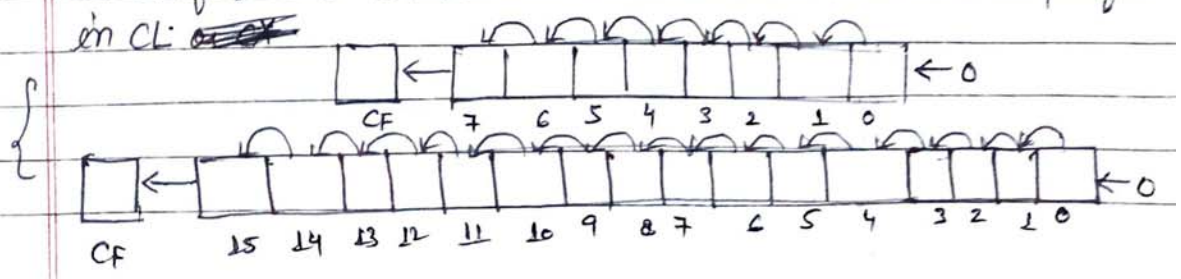
(ii) XOR mem, data eg:- XOR START, 05H

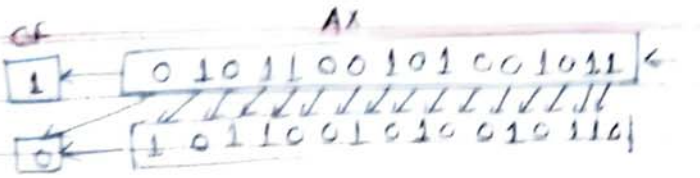
(iii) XOR reg, data eg:- XOR AL, 05H

(iv) XOR A, data eg:- XOR AX, 1234H

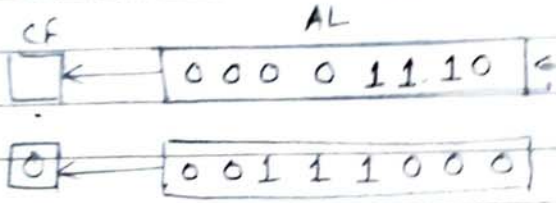
(v) SHL/SAL: These instructions perform shift operations and shift the operand byte bit by bit to the left and insert zeros in the newly introduced least significant bits.

In Shift/Rotate instructions the count is either 1 or specified in CL:





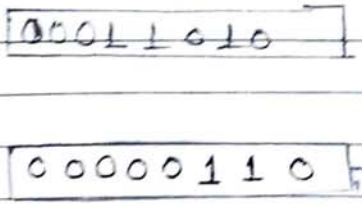
SHL AL, CL ; AL = 0E, CL = 02H



NOTE: SAL instruction can be used instead of SHL.

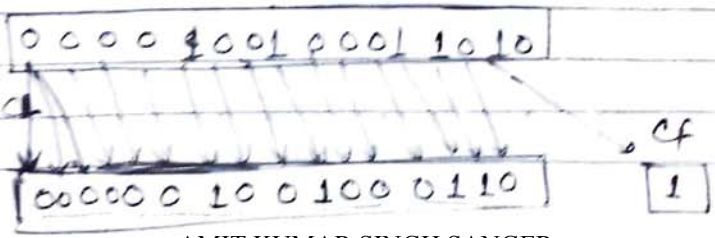
(vi) SHR: This instruction performs bit wise right shifts on operand word or byte that may reside in a register or a memory location by the specified count in the instruction and inserts zeros in the shifted portions.
This instruction shift the operand through carry.

eg: DH = 1AH CL = 02H

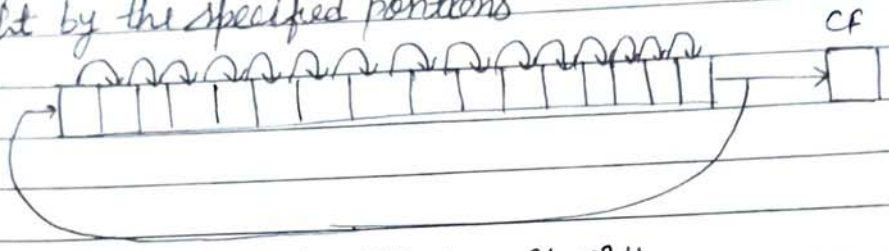


(vii) SAR: This instruction shifts the operand word or byte by the number of bit positions equal to count and fill the vacant positions on the left with original most significant bit.

eg: AX = 091AH, CL = 02H



(viii) ROR: This instruction rotates the contents of the destination operand to the right either by one or by the count specified in CL, excluding carry.
The least significant bit is pushed into the carry flag and simultaneously it is transferred into the most significant bit position at each operation. The remaining bits are shifted right by the specified positions.



eg:- BX = AF5DH CL = 2H

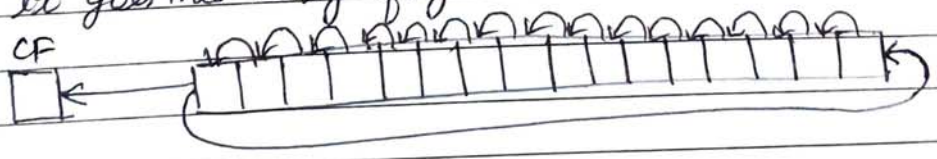
Initial: 1010 1111 0101 1101 CF 1

RoR BX, CL

CL=1 1101 0111 1010 1110 CF 0

CL=2 0110 1011 1101 0111

(ix) ROL: ~~Rotates~~ Rotates the operand left by the number of bit positions equal to count. Each bit is shifted out from left most bit goes back into rightmost bit position; also it goes into carry flag.



eg:- BX = AF5DH, CL = 2H

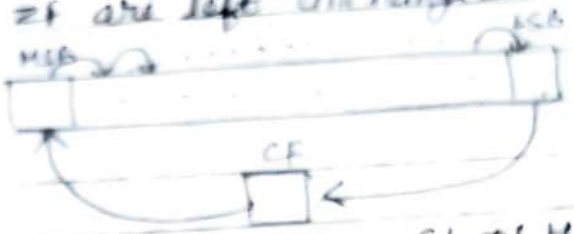
1010 1111 0101 1101 BX

ROL BX, CL

CL=1 CF 1 ← 0101 1110 1011 1011

CL=2 0 1011 1101 0111 0110

(x) **RCR** This instruction rotates the contents of operand right by count specified through carry for each operation carry flag is pushed into MSB and LSB is pushed into carry flag
The SF, PF, ZF are left unchanged



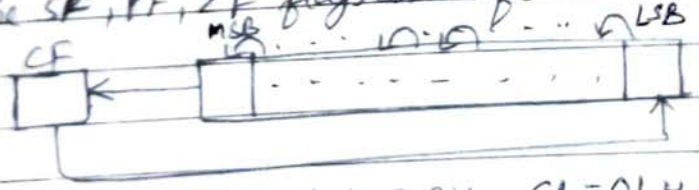
eg: BX = 1AFBH, CL = 0LH

RCR BX, CL

CF: 0 | 0001 1010 1111 1011

After execution: CF: 1 | 0000 1101 0111 1101

(XI) **RCL**: Rotates the contents of operand left by specified count through carry flag. For each operation, the carry flag is pushed into LSB and MSB of operand is pushed into carry flag. The remaining bits shifted left by specified position.
The SF, PF, ZF flags are left unchanged.



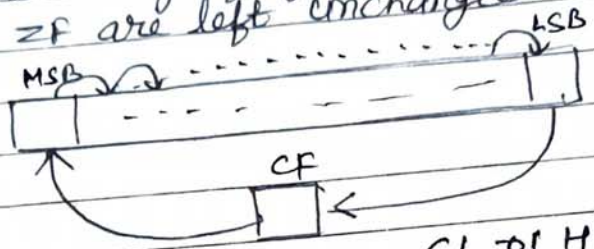
eg: BX = 1AFBH CL = 0LH

RCL BX, CL

CF: 0 | 0001 1010 1111 1011

After execution: CF: 0 | 0011 0101 1111 0110

(X) RCR: This instruction rotates the contents of operand right by count specified through carry. For each operation carry flag is pushed into MSB and LSB is pushed into carry flag.
The SF, PF, ZF are left unchanged.



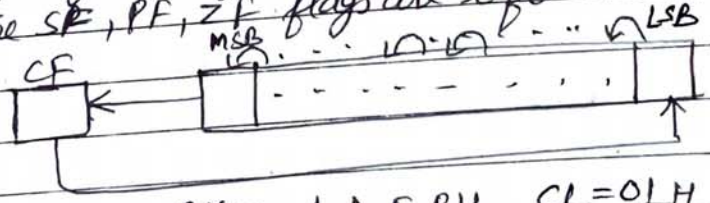
eg:- BX = 1AFBH, CL = 01H

RCR BX, CL

CF = 0 | 0001 1010 1111 1011

After execution CF = 1 | 0000 1101 0111 1101

(Xi) RCL: Rotates the contents of operand left by specified count through carry flag. For each operation, the carry flag is pushed into LSB and MSB of operand is pushed into carry flag. The remaining bits shifted left by specified position.
The SF, PF, ZF flags are left unchanged.



eg:- BX = 1AFBH CL = 01H

RCL BX, CL

CF = 0 | 0001 1010 1111 1011

After execution CF = 0 | 0011 0101 1111 0110

4. Compare Instruction: This instruction compares a byte/word in the specified source with a byte or word in the specified destination. The source can be an immediate number, register or memory location. The destination can be a register or a memory location.

- Source and destination both can not be memory locations.
- The comparision is actually done by subtracting the source byte/word from destination byte or word.
- The flags are set to indicate the results of the comparision. AF, OF, SF, ZF, PF, CF are updated by the CMP instruction.

Syntax: CMP destination, source

	CF	ZF	SF
destination = source	0	1	0
destination > source	0	0	0
destination < source	1	0	1

eg: CMP AL, 05H
CMP BH, CL

5. Transfer of control instructions: The transfer control instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP register get loaded with new values. This type of instructions are classified in two types:

(i) Unconditional control transfer instruction: In case of unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified.

(ii) Conditional control transfer instructions: In the conditional control transfer instructions, the control is transferred to the specified location provided the result of the previous operation satisfies a particular condition, otherwise, the execution continues in normal flow sequence. The results of the previous operations are replicated by condition code flag.

Unconditional Branch Instructions

- JMP
- CALL
- RET
- IRET
- INT N
- INTO
- LOOP

• JMP : (Unconditional Jump): This instruction unconditionally transfer the control of execution to the specified address using a displacement (8 bit / 16 bit).
No flags are affected by this instruction.

JMP	Displacement 8bit		Intrasegment, relative, near jump
JMP	Displacement 16 bit	Displacement 16 bit	Intersegment, relative far jump
JMP	IP (LB) IP (UB)	CS (LB) CS (UB)	Intersegment direct

eg:

JMP BX

BX [0010H]

• CALL : (Unconditional call): This instruction is used to call a subroutine / procedure from a main program. The address of the procedure may specify directly or indirectly depending on the address mode.
There are two types of procedures depending on whether it is available in the same segment or in another segment. The modes for them are respectively called as intrasegment and intersegment addressing. On execution, this instruction stores the incremented IP and CS onto the stack and loads the CS and IP registers, respectively with the segment and offset address of the procedure to be called.

• RET (Return from the procedure)
At each CALL instruction, the IP and CS of the next instruction is pushed onto stack, before the control is transferred to the procedure. At the end of the procedure, the RET instructions must be executed. When it is executed, the previously stored content of IP and CS along with flags are retrieved into the CS, IP and flag registers from the stack and the execution of main program continues further.
In case of a FAR procedure the current content of SP points to IP and CS at the time of return. While in case of a NEAR procedure, it points to only IP. Depending upon the type of procedure and the SP contents, the RET instruction is of four types.

1. Return within segment
2. Return within segment adding 16-bit immediate displacement to the SP contents.
3. Return intersegment.
4. Return intersegment adding 16-bit immediate displacement to the SP contents.

• IRET (Return from ISR)

When an interrupt service routine is to be called, before transferring control to it, the IP, CS and flag registers are stored onto the stack to indicate the location from where the execution is to be continued after the ISR is executed. So, at the end of each ISR is executed, the value of IP, CS, and flags are retrieved from the stack to continue the execution of the main program. The stack is modified accordingly.

• INT N (Interrupt Type N): In the interrupt structure of 8086/8088, 256 interrupts are defined corresponding to the types from 00H to FFH. When an INT N instruction is executed, the TYPE byte N is multiplied by 4 and the contents of IP and CS of interrupt service routine will be taken from the hexadecimal multiplication as offset address and 0000 as segment address.

eg:- INT 20H

Type X4 = 20X4 = 80H, Pointer IP and CS of ISR is 0000H: 0080H

• INTO (Interrupt on overflow): This command is executed when the overflow flag OF is set. The new contents of IP and CS are taken from the address 0000:0010 as explained in INT type instruction. This is equivalent to type 4 instruction.

• LOOP (Loop unconditionally): This instruction executes the part of the program from label or address specified in the instruction up to the loop instruction; CX number of times.

At each iteration, CX is decremented automatically.

eg:
```
MOV CX, 0005H
MOV BX, OFF7H
label: MOV AX, CODE1
OR BX, AX
AND DX, AX
LOOP label
```

Conditional Branch instruction

- JCC
- LOOPE/LOOPZ
- LOOPNE/LOOPNZ

- JCC : (Jump Conditional)

When these instructions are executed, they transfer execution control to the address specified relatively in instruction, provided the condition in the opcode is satisfied; otherwise, the execution continues sequentially.

The condition means the status of condition code flags.

The address has to be specified in the instruction relatively in terms of displacement, which must lie within -80H to 7FH bytes from the address of the branch instruction.

In other words, only short jumps can be implemented using conditional branch instructions.

A label may represent the displacement.

- Following are the conditional branch instructions:

S.No	Mnemonics	Displacement	operation
1.	JZ/JE	label	Transfer execution control to address label if ZF=1
2.	JNZ/JNE	label	Transfer execution control to address label if ZF=0
3.	JS	label	Transfer execution control to address label if SF=1
4.	JNS	label	Transfer execution control to address label if SF=0
5.	JO	label	Transfer execution control to address label if OF=1
6.	JNO	label	Transfer execution control to address label if OF=0
7.	JP/JPE	label	Transfer execution control to address label if PF=1
8.	JNP	label	Transfer execution control to address label if PF=0
9.	JB/JNAE/JC	label	Transfer execution control to address label if CF=1
10.	JNB/JAE/JNC	label	Transfer execution control to address label if CF=0

S.NO	Mnemonics	Displacement	operation
11.	JBE / JNA	label	Transfer execution control to address label if CF = 1 or ZF = 1.
12.	JNBE / JA	label	Transfer execution control to address label if CF = 0 or ZF = 0.
13.	JL / JNGE	label	Transfer execution control to address label if neither SF = 1 nor OF = 1.
14.	JNL / JGE	label	Transfer execution control to address label if neither SF = 0 nor OF = 0.
15.	JLE / JNC	label	Transfer execution control to address label if ZF = 1 or neither SF nor OF is
16.	JNLE / JE	label	Transfer execution control to address label if ZF = 0 or at least any one of SF and OF is 1.

· LOOPZ / LOOPE, LOOPNZ / LOOPENE:

The conditional LOOP instructions are used to implement structures like DO_WHILE, REPEAT_UNTIL, etc.

Mnemonics	Displacement	Operation
LOOPZ / LOOPE	Label	Loop through a sequence of instructions from label while ZF = 1 and CX is not 0.
LOOPNZ / LOOPENE	label	Loop through a sequence of instructions from label while ZF = 0 and CX is not 0.

6. Flag Manipulation and Processor Control instructions:

These instructions control the functioning of the available hardware inside the processor chip.

There are categorized into two types:

(a.) Flag manipulation instruction

(b.) Machine control instruction

The flag manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution.

The flag manipulation instructions are
CLC - Clear Carry flag.
CMC - Complement Carry flag
STC - Set carry flag
CLD - Clear Direction flag
STD - Set Direction flag
CLI - Clear Interrupt flag
STI - Set Interrupt flag

- These instructions modify CF, DF, IF flags directly.
- The DF and IF flags may be modified using the flag manipulation instructions, further control the processor operations. Thus respective instructions may also be called machine or process control instructions.
- The machine control instructions supported by 8086 are;
WAIT - Wait for Test input pin to go low.
HLT - Halt the processor.
NOP - No operation
ESC - Escape to external device like NDP.
LOCK - Bus lock instruction prefix.

7. String Manipulation Instructions: A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually, are called as byte string or word string.
• REP • MOVSB/MOVSW • CMPSB/CMPSW
• SCASB/SCASW • STOSB/STOSW • LODSB/LODSW
• REP (Repeat Instruction Prefix): This instruction is used as a prefix to other instructions. The instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero, when CX becomes zero, the execution proceeds to the next instruction in sequence. There are two more options of the RPE instruction. The first is REPE/REPZ. The second is REPNE/REPNZ allows for repeating the operation which not equal/not zero.

These options are used for CMPS, SCAS instructions only as instruction prefixes.

• MOVS / MOVSB / MOVSW (Move string byte or string word):
This instruction copies a byte or a word from location in the data segment to a location in the extra segment. The offset of the source in the data segment must be in the SI register. The offset of the destination in the extra segment must be in the DI register. For multiple byte or multiple word moves, the number of elements to be moved is put in the CX register so that it can function as a counter. After the byte or a word is moved, SI and DI are automatically adjusted to point to the next source element and the next destination element. If DF is 0, then SI and DI will be incremented by 1 or 2 after a byte or word move. If DF is 1 then SI and DI will be decremented by 1 or 2 after a byte or word move. MOVS does not affect any flag.

eg:-
```
MOV SI, OFFSET SOURCE
MOV DI, OFFSET DESTINATION
CLD
MOV CX, 04H
REP MOVSB
```

• CMPS / CMPSB / CMPSW (Compare String byte or String word):
The CMPS instruction can be used to compare two strings of bytes or words. The length of string must be stored in the register CX. If both bytes or words strings are equal, zero flag is set. The flags are affected in the same ways as CMP instruction. The DS:SI and ES:DI point to the two strings. The REP instruction prefix is used to repeat the operation till CX becomes zero or the condition specified by the REP prefix is false.

eg:-
```
MOV AX, SEG1
MOV DS, AX
MOV AX, SEG2
MOV ES, AX
MOV SI, OFFSET STRING1
MOV DI, OFFSET STRING2
MOV CX, 010H
CLD
REPE CMPSW
```

• SCAS / SCASB / SCASW : (Scan string Byte or string word) :

- This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX.
- The string is pointed to by ES : DI register pair. The length of string is stored in CX. The DF controls the mode for scanning of the string.
- Whenever a match is found in the string, execution stops and the zero flag is set. If no match is found the zero flag is reset.

The REPNE prefix is used with the SCAS instruction. The pointers and counters are updated automatically, till a match is found.

eg: MOV DI, OFFSET STRING
MOV AL, 0DH
MOV CX, 80
CLD
REPNE SCAS STRING

• LODS / LODSB / LODSW (Load String Byte or String word) :

The LODS instruction loads the AL/AX register by the content of a string pointed to by DS : SI register pair. The SI is modified automatically depending upon DF.

If it is a byte transfer the SI is modified by one and if it is a word transfer, the SI is modified by two.

No other flags are affected.

Example:
CLD
MOV SI, OFFSET SOURCE
LODS SOURCE

• STOS / STOSB / STOSW (Store string Byte or String word) : The STOS instruction stores the AL/AX register contents to a location in the string pointed by ES : DI register pair. The DI is modified accordingly. No flags are affected by this instruction.

The DF flag controls the string instruction execution. SI and DI are modified after each iteration automatically. If DF = 0 execution follows autoincrement mode. If DF = 1 execution follows autodecrement mode.

eg: MOV DI, OFFSET TARGET
STOSB

INTERRUPTS: Software and Hardware

The meaning of 'interrupts' is to break the sequence of operation. While the CPU is executing a program, an 'interrupt' breaks the normal sequence of execution of instructions, diverts its execution to some other program called interrupt service routine (ISR). After executing ISR, the control is transferred back again to the main program.

Whenever a number of devices interrupt a CPU at a time, and if the processor is able to handle them properly, it is said to have multiple interrupt processing capability.

Need for interrupt: Interrupts are particularly useful when interfacing I/O devices that provide or require data at relatively low data transfer rate

Source of interrupt in 8086 microprocessor:

An 8086 interrupt can come from any one of the three sources:

1. **External Signal:** An 8086 can get interrupt from an external signal applied to the non maskable interrupt (NMI) input pin, or the interrupt (INTR) input pin.
2. **Special Instruction:** An execution of the interrupt instruction (INT). This is referred as software interrupt.
3. **Condition produced by instruction:** An 8086 is interrupted by some condition produced in the program by the execution of an instruction. For example, divide by zero.

Types of Interrupts:

There are two types of interrupts in 8086

(i) Hardware interrupts - INTR, NMI

(ii) Software interrupts - INT n

(i) **Hardware Interrupts:** The intel microprocessors support hardware interrupts through two pins that allow interrupt requests INTR and NMI. One pin is also there to acknowledge, $\overline{INTA}$, the interrupt request INTR.

- **Markable Interrupt (INTR):**
  - → INTR pin is provided in 8086 microprocessor.
  - → It has lower priority than NMI (Non Markable Interrupts).

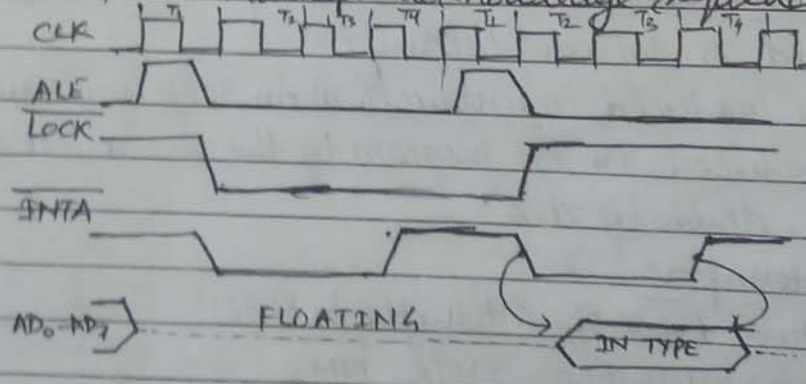
→ The priorities within the INTR are decided by the type of the INTR signal that is to be passed to the processor via data bus external device.

→ INTR signal is level triggered and can be masked.
  ⇒ If IF flag is set, the INTR interrupt is enabled.
  ⇒ If IF flag is cleared, the INTR is disabled.

→ INTR is activated by a high level signal in the INTR pins. If the INTR pin is activated in the last clock cycle of current instruction or before that then it will be responded after the execution of current instruction.

→ If IF flag is set and INTR is activated then processor responds to the INTR signal and it automatically clears IF for two reasons
  - To prevent a signal on the INTR input from interrupting a higher priority ISR in progress.
  - To make sure that a single signal in the high state, existing for a sufficient duration does not cause 8086 to interrupt again before completing the execution of ISR.

→ The 8086 does two interrupt acknowledge cycle when it receives the INTR. The interrupt acknowledge sequence is as



• Non-Maskable Interrupt

→ 8086 processor has a non maskable interrupt input pin (NMI).

→ It has highest priority among the external interrupts. TRAP (Type 1) is an internal interrupt having the highest priority amongst all the interrupt except the Divide by Zero (Type 0) exception.

→ NMI is activated on a positive transition.

→ NMI interrupt is equivalent to instruction INT 02.

→ When NMI is activated the current instruction being executed is completed and then the NMI is served.

(ii) Software Interrupt:

→ Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers.

→ It includes -

INT - Interrupt instruction with type number.

It is 2 byte instruction. First byte provides the op code and the second byte provides the interrupt type number.

→ Its execution includes the following steps -

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location 'type number X4'.
- CS is loaded from the contents of the next word location.
- Interrupt flag and Trap flag are reset to 0.

→ There are 256 interrupt types under this group.

- INT 00H, INT 01H, ..., INT FFH

In 8086 the memory location to which an interrupt goes is always four times the value of the interrupt number.

256 interrupts are divided into 3 groups:

⇒ Type 0 to Type 4 interrupts: These are used for fixed operations and hence are called dedicated interrupts.

⇒ Type 5 to Type 31 interrupts: Not used by 8086, reserved by Intel for higher processors like 80286, 80386, etc.

⇒ Type 32 to 255 interrupts: Available for users, called user defined interrupts. These can be hardware interrupts and activated through INTR line or can be software interrupts.

The lowest five interrupts types in the 8086 are dedicated to specific interrupts

- Type 00H or Divide by Zero interrupt
  - If the quotient from a DIV or IDIV operation is too large to fit into the result register.
  - If the divisor is very small compared to the dividend.
  - 8086 automatically generates a Type 0 interrupt.
- Type 01H, Single step or TRAP interrupt
  - If trap flag is set 8086 will do a type 1 interrupt after every instruction execution.
- Type 02H, or NMI interrupt
  - When 8086 receives a low to high transition on its NMI input.
  - Type 02H interrupt response cannot be disabled by any program instruction.
  - Could be used for handling any critical situations like power failure detection.
- Type 03H or One byte INT interrupt or Break point interrupt
  - INT 3H instruction - to implement break point routines.
  - The system execute instruction up to break point and then goes to break point routine. Used for debugging.
- Type 04H or Overflow interrupt
  - INTO: Interrupt on overflow instruction used for invoking an interrupt after overflow in an arithmetic operation. If no overflow it will be a NOP instruction.
  - There are two ways to detect and respond to an overflow error in a program:
    - ⇒ Place the jump on overflow (JO) instruction immediately after the arithmetic instruction.
    - ⇒ Place the INTO instruction immediately after the arithmetic instruction in program.

Interrupt Priority: There is following interrupt priority structure in 8086 microprocessor:

1. Divide error, INT nn, INTO
2. NMI
3. INTR
4. Single Step.

High ↓ Low

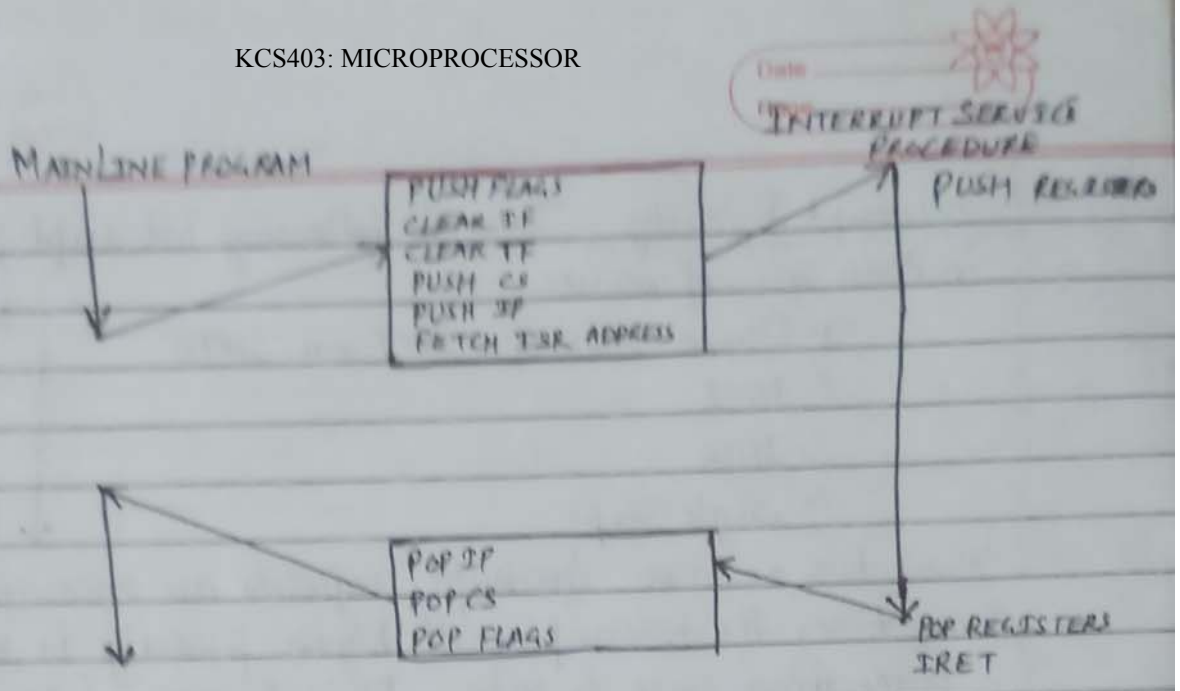
When two or more interrupt requests are encountered at the same time, the interrupt with higher priority is serviced first.

Interesting case is when Divide by zero interrupt and NMI occur at same time.

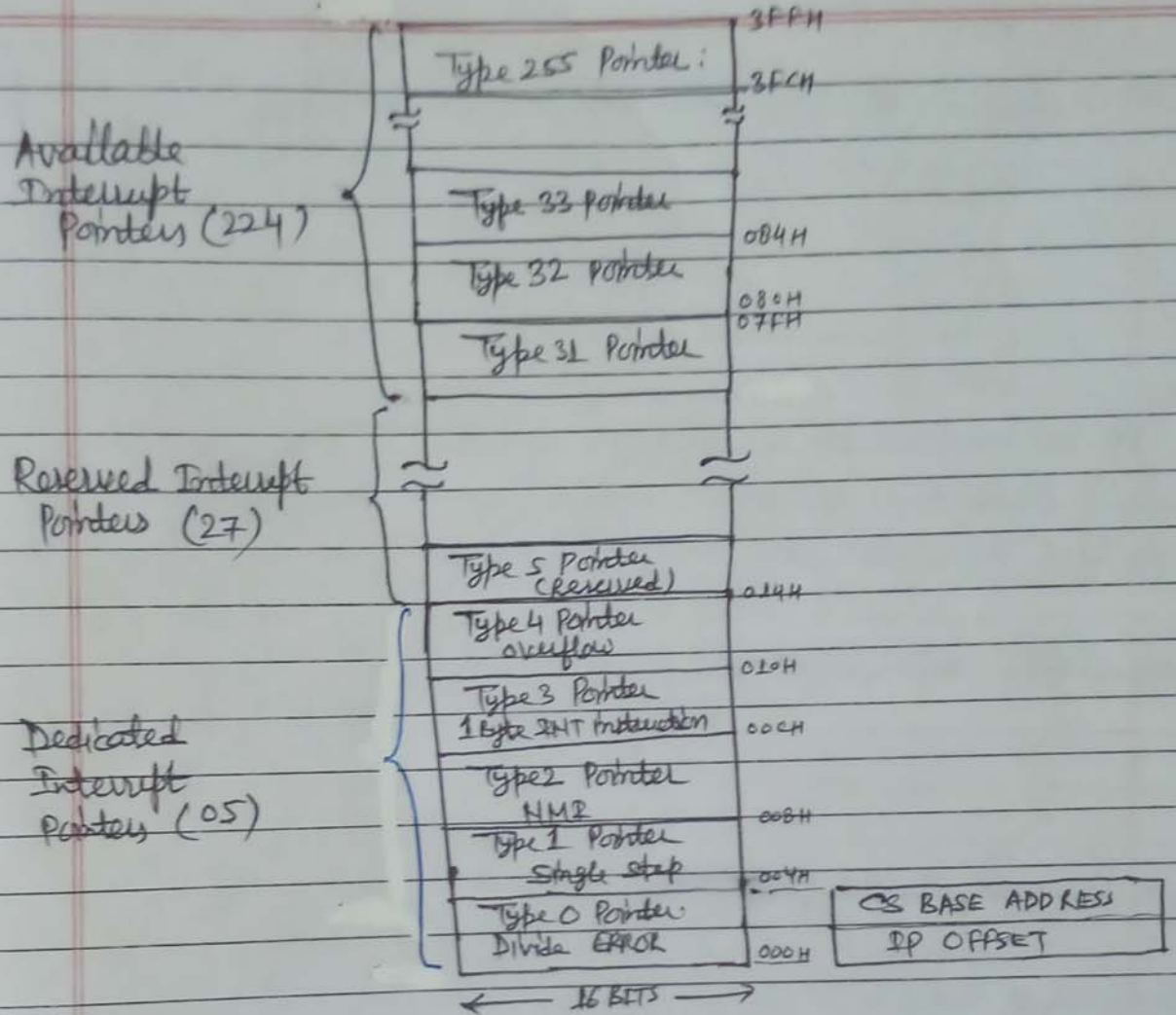
Since the 8086 checks for internal interrupts before checks for NMI, the Divide by zero is taken first.

8086 Interrupt Processing:

- At the end of each instruction cycle, 8086 checks to see if any interrupts have been requested.
- If an interrupt has been requested, the 8086 responds to interrupt by stepping through the following series of major steps:
  1. It decrements the stack pointer by 2 pushes the flag register on the stack.
  2. It disables the 8086 INTR interrupt input by clearing the IF in the flag register.
  3. It resets TF (Trap Flag) in the flag register.
  4. It decrements the stack pointer by 2 and pushes the current code segment register contents on the stack.
  5. It decrements the stack pointer again by 2 and pushes the current instruction pointer contents on the stack.
  6. It does an indirect far jump to start of the procedure by loading the CS and IP values for the start of the interrupt service routine.
- An IRET instruction at the end of the interrupt service routine returns execution to main program.



- The 8086 gets the new values of CS and IP registers from four memory addresses.
- When it responds to an interrupt, the 8086 goes to memory locations to get the CS and IP values to start of the interrupt service routine.
- In an 8086 system the first 1 kB of memory from 00000H to 003FFH is reserved for storing the starting address of interrupt service routine.
- This block of memory is often called the INTERRUPT VECTOR TABLE or the INTERRUPT POINTER TABLE.
- Since 4 bytes are required to store the CS and IP values for each interrupt service procedure, the table can have the starting address for 256 interrupt service routines.
- Each interrupt type is given a number between 0 and 255 and the address of each interrup is found by multiplying the type by 4.
- When 8086 responds to an interrupt, it automatically goes to specified location in the interrupt vector table to get the starting address of interrupt service routine.



Interrupt Vector Table