mood-book

PART-1 : 8086 ARCHITECTURE

Introduction:

Microprocessor:

A Microprocessor is an integrated circuit (IC) which incorporates core functions of a computer's central processing unit (CPU). It is a programmable multipurpose Silicon chip, clock driven, register based, accepts binary data as input and provides output after processing it as per the instructions stored in the memory.

Examples:

Intel 4004, 8085, 8086, pentium 4, Core i3, Core i5 core i7 etc.

4004 → first 4-bit Microprocessor designed in 1971
8008 → first 8-bit Microprocessor designed in 1972
8080 → general purpose 8-bit Microprocessor.
8085 → functionally complete 8-bit Microprocessor designed in 1977
8086 → first 16-bit Microprocessor from Intel.

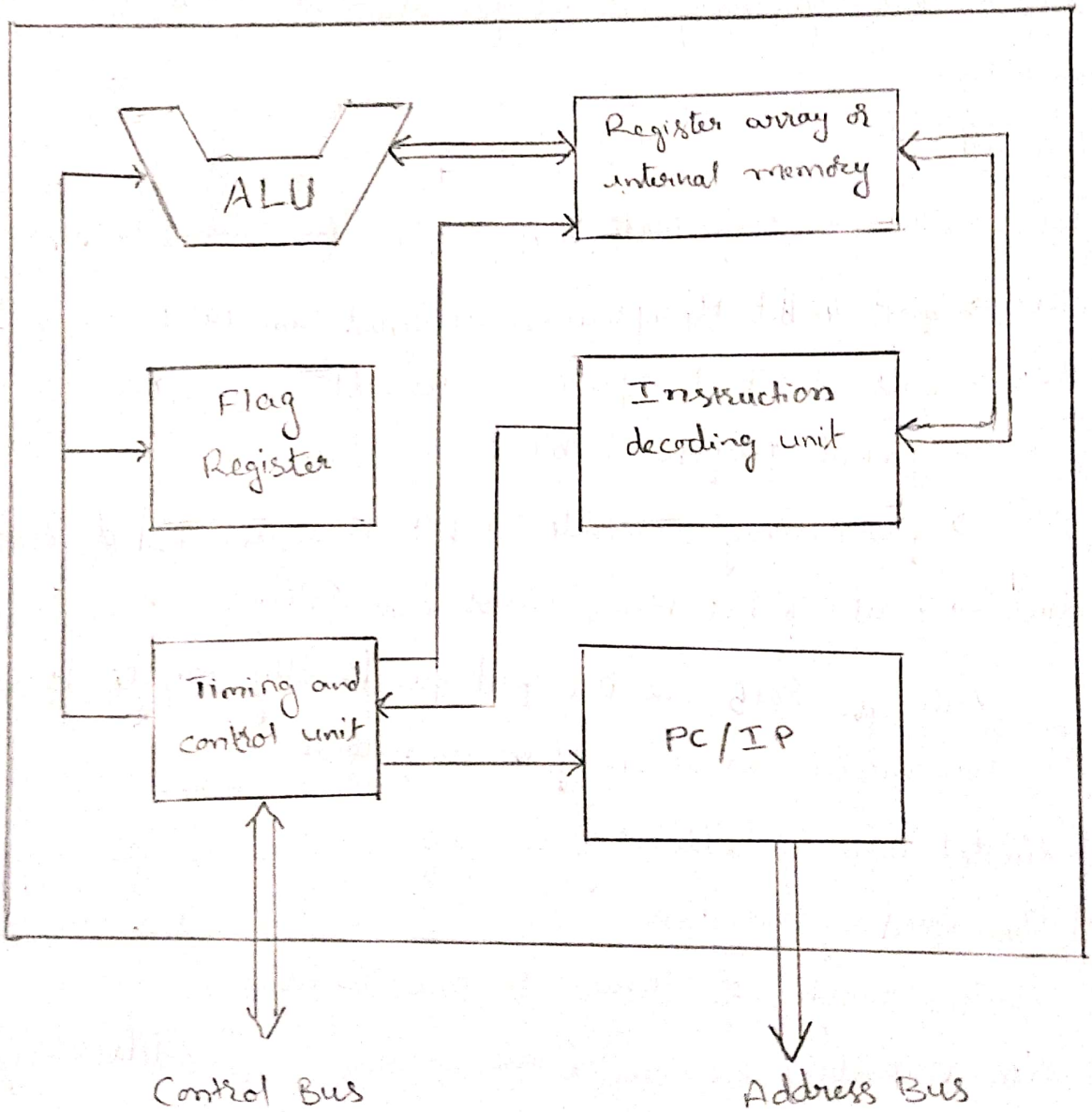
Although 8085 was the first functionally complete Microprocessor It has various limitations ~~of 8086~~ as follows.

1) Limited memory addressing capacity
2) Slow speed of execution
3) Limited number of general purpose registers
4) Non-availability of complex instructions and addressing modes
5) It doesnot support pipelining (or) parallelism

8086 Microprocessor:

The first 16-bit Microprocessor from Intel (1978) which contains a set of 16-bit general purpose registers., supports a 16-bit ALU, a rich instruction set and a segmented memory-addressing scheme.

Functional Block diagram of a microprocessor:



The functional Block diagram of a microprocessor is shown and it consists of various Blocks such as

(i) ALU
(ii) Register array
(iii) flag register
(iv) Timing and control unit
(v) Instruction decoding unit
(vi) PC/IP (program counter/instruction pointer)

ALU: It is the computational unit which performs arithmetic & Logic operations.

Register array: Internal storage of data which is processed by the Microprocessor is stored in Register array.

Flag register: Various conditions of the results are stored as status bits called flags in flag register.

Timing and control unit: It generates various control signals for internal and external operations of the Microprocessor.

Instruction decoding unit: It decodes instructions; and sends the information to the timing and control unit.

PC/IP: It generates the address of the instructions to be fetched from the memory and send it through address bus to the memor

Register Organization of 8086:

8086 has a powerful set of registers known as general purpose and special purpose registers. All of them are 16-bit registers. The general purpose registers can be used as either 8-bit registers or 16-bit registers.

They may be either used for holding data, variables & intermedi results temporarily or for other purposes like a counter etc. or addressing modes etc.

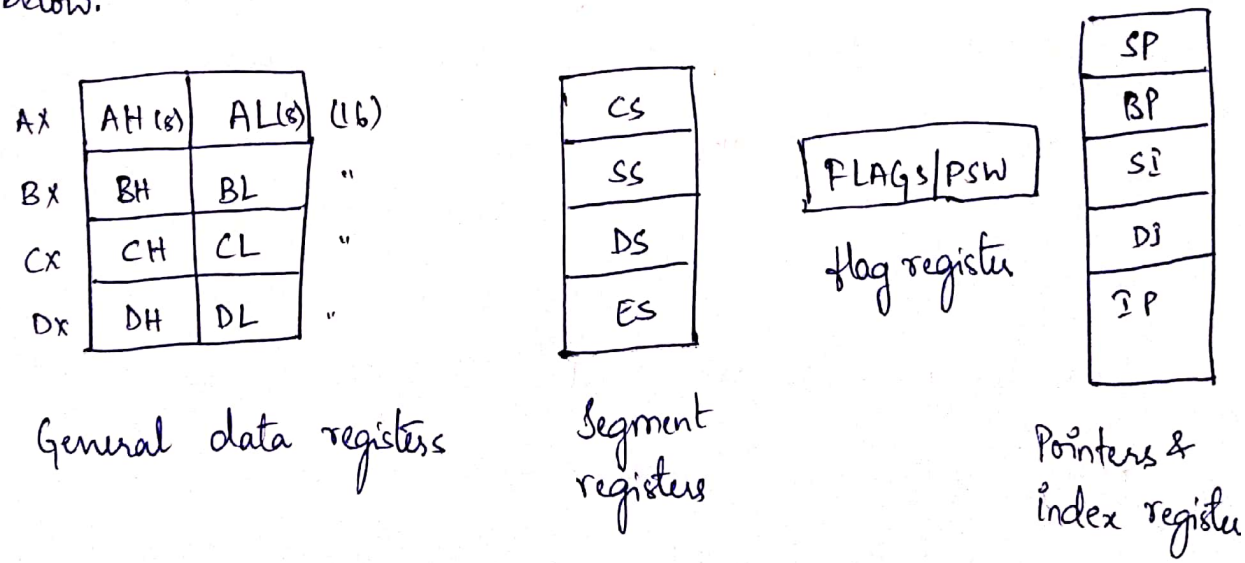
The Special purpose register are used as Segment registers, Pointers, index registers or as an offset storage registers for Particular addressing modes.

The registers can be classified into four groups

(i) General purpose data registers
(ii) Segment registers
(iii) pointer and index registers.
(iv) Flag register.

General Data registers:

The registers organisation of the 8086 Microprocessor is as shown below.



Ax: Ax is used as 16-bit with lower 8-bits of Ax designated as AL and higher 8 bits as AH. AL can be used as an 8 bit accumulator for 8 bit operations.

Bx: The register Bx is used as an offset storage for forming physical addresses in case of certain addressing mode. c.

Cx: The register Cx is also used as a default Counter in case of string and loop instructions.

Dx: Dx registers is a general purpose register which may be used as an implicit operand or destination in case of a few instruction

Segment registers:

The 8086 addresses a segmented memory. The Total memory that an 8086 μp has is 1 Mega Byte memory. The entire memory is divided into 16 logical segments. Each segment which is divided will be of 64K bytes of memory.

There are four segment registers as follows

1) Code Segment register
2) Data Segment register
3) Extra Segment register
4) Stack Segment register

The Code Segment register is used for addressing a memory location in the Code Segment of the memory, where the Executall Program is stored.

The CS contains the base or start of the current code Segment : Ip contains the distance or offset address from this address to the next instruction byte to be fetched.

The BIU computes the 20-bit physical address by logically shifting the contents of cs 4-bits to the left and adding the 16-bit contents of Ip.

Data Segment register : It points to the current data segment. Th operands for most instructions are fetched from this segment. The 16-bit contents of the source index (SI) or destination Ind are used as offset for computing 20-bit physical addres

Extra Segment register : It points to the extra segments in which data in excess of 64K pointed by the DS is stored. String instructions use the ES & DI to determine the 20-bit physical address for the destination.

Stack Segment register : It points to the current stack. The 20 bit physical stack address is calculated from the stack Segmen & the stack pointer (sp) for stack instructions such as push & p

While addressing any location in the memory bank, the physical address is calculated from two parts, the first is Segment address & second is offset. The Segments registers contain 16 bit Segment base addresses related to different Segment

Pointers and Index Registers :

The pointers contain offset within the particular segments. The Pointers IP, BP, & SP usually contains offset within the Code & stack Segments. Th

The Index registers are used as general purpose registers as well as for offset storage in case of indexed, based indexed and relative based indexed indexed addressing modes.

The register SI is generally used to store the offset of source data in data Segment while the DI is the offset for destination in extra or data Segment. There are generally used in string Manipulation instructions

Flag register:

8086 has a 16 bit flag register which is divided into two parts which are

1) Conditional or status flag
2) Machine Control flags.

The condition flag is the lower byte of the 16-bit flag register and the control flag register is the higher byte of the register.

The bit configuration of 8086 flag register is as follows.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	O	D	I	T	S	Z	X	Ac	X	P	X	C_y

O - Overflow flag
D - Direction flag
I - Interrupt flag
T - Trap flag
S - Sign flag
Z - Zero flag
Ac - Auxillary Carry flag
P - Parity flag
C_y - Carry flag
X - Not used.

The description of each flag bit is as follows.

Status flags:

Carry flag: It is set when there is a Carry out of the MSB or a borrow in case of subtraction.

Eg: AL = 98H = 1001 1000
BL = 87H 1000 0111
1 0001 1111

∴ There Carry flag is set to 1

Parity flag: This flag is set to 1 if the lower byte of the result contains even number of 1s

Eg: AH = 48H : 0100 1000
BH = 62H : 0110 0000
1010 1000

As the lower byte of the result has odd number of 1's the Parity flag is set to 0

Auxiliary Carry flag: It is set to 1, if there is any Carry from lower nibble to higher nibble or borrow from higher nibble to lower nibble otherwise it is set to Zero.

Eg: AL = A8H : 1010 1000
BL = 99H 1001 1001
1 0100 0001

∴ the A_C is set to 1 as there is a Carry from lower nibble

~~Zero flag: This flag is set to 0 if the~~ .

Zero flag: This flag is set if the result of the computation or comparision performed by the instructions is Zero.

Sign flag: This flag is set when the result of any computation is negative. For Signed computations, the Sign flag equals the MSB of the result.

Overflow flag: This flag is set if an overflow occurs i.e. if the result of a Signed operation is large enough to be accommodated in a destination register. otherwise it is set to ze

Eg: AL = 98H → 1001 1000
BL = C9H → 1100 1001
1 0110 0001

∴ The overflow flag is set to 1 .

Control flags:

Trap flag: If this flag is set, the processor enters the single step execution mode. In other words, a trap interrupt is generated after execution of each instruction. The processor executes the current instruction and the control is transferred to the trap interrupt service routine.

Interrupt flag: If this flag is set, the maskable interrupts are recognised by the CPU, otherwise they are ignored.

Direction flag: This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address ie: auto incrementing mode. otherwise, the string is processed from the higher address towards the lowest address ie auto decrementing mode.

Memory Segmentation:

The Memory of an 8086 µp is organised as segmented memory. The entire memory which is physically available will be divided into a number of logical segments.

8086 µp has 20 bit address bus that means it can access 2^{20} memory locations which is 1MB of memory is available for 8086 µp.

The Total 1MB of physical memory is divided into 16 logical segments, Each of 64Kbytes size.

The maximum size of a segment is 64KB (2^{16}). and minimum size of the segment is 16 Bytes.

To access any particular location in the memory, the Processor needs to have the knowledge of the actual physical address. The 8086 µP has an address but of 20 bits it is not possible to store 20 bit address in 8086 µP. And hence the physical address should be stored in terms of two 16 bit registers.

The 16 bit contents of the Segment register point to the starting location of a particular segment and in the segment if to address a specific memory location a 16 bit offset address must be used.

The addresses in the Segment register can be assigned from 0000H to F000H respectively. The offset address values are from 0000H to FFFFH such that the physical address range from 00000 to FFFFF.

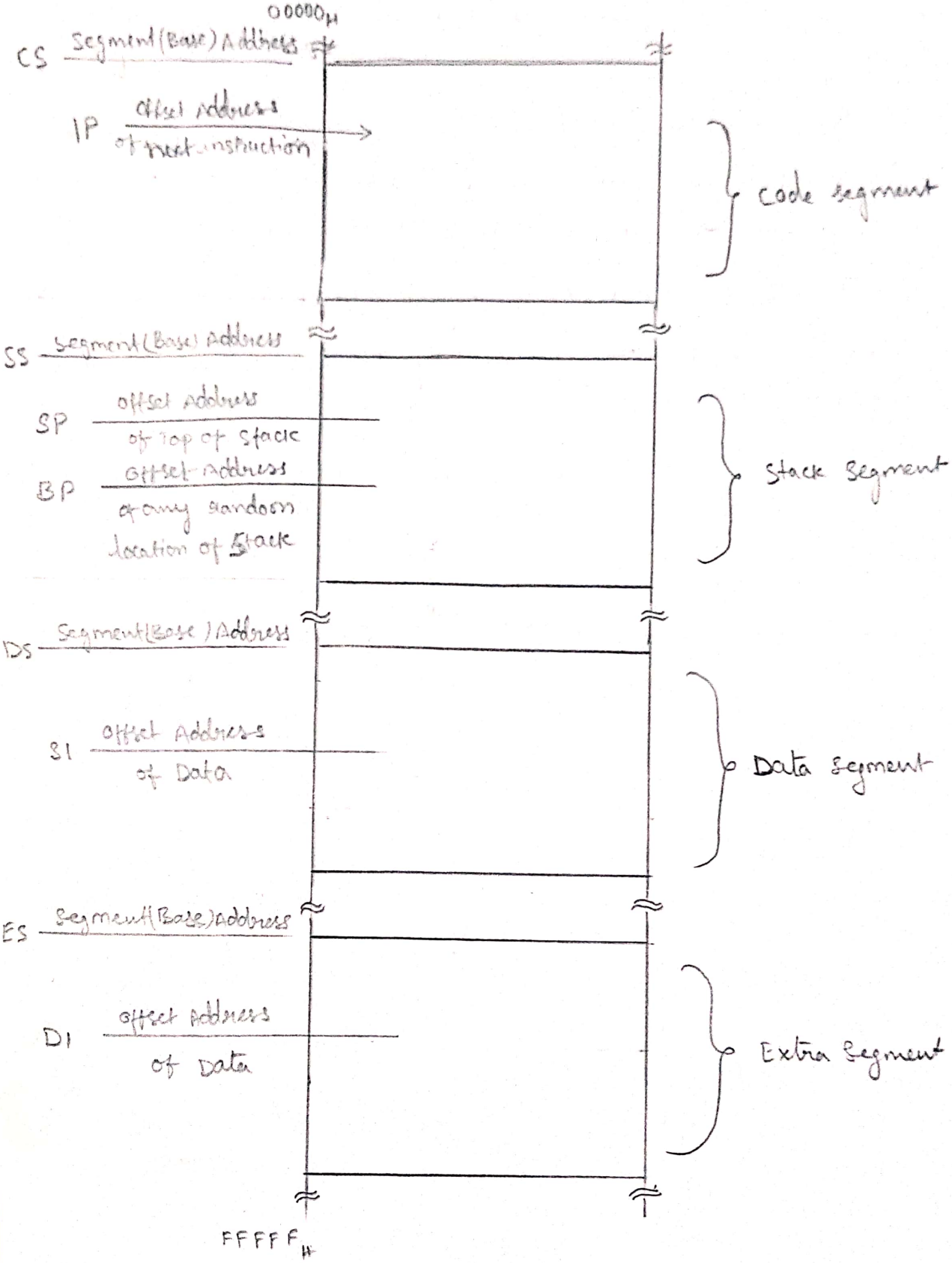
The diffrent types of logical segments are

1) Code Segment
2) Stack Segment
3) Data Segment
4) Extra Segment.

These Segments can be implemented in two ways.

1) overlapping Segments
2) Non overlapping Segments. } (fig ②)

If a Segment starts at a particular address and has a maximum size of 64 K Bytes. If another Segments starts before this 64 K bytes of the first segment then the two



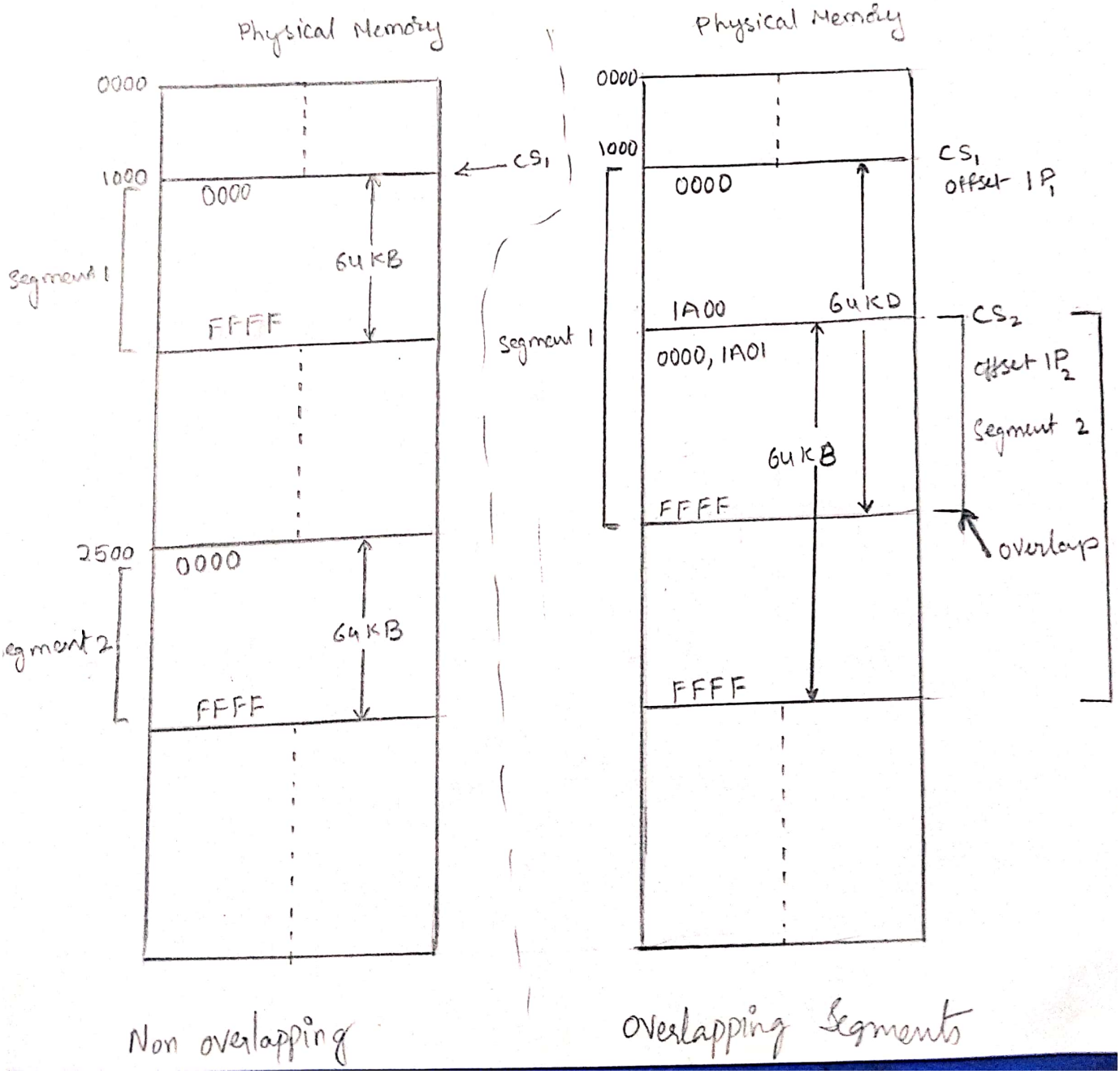
are said to be overlapping segments and if they are independent to each other then they are called as non-overlapping segments.

The area of memory from start of the second segment to the possible end of the first segment is called as an overlapped segment area.

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

In the overlapped area physical address = $CS_1 + IP_1 = CS_2 + IP_2$

where '+' indicates the procedure of physical address formation



Advantages of memory Segmentation:

The main advantages of memory Segmentation are

1) Allows the memory Capacity to be 1MBytes although the actual addresses to be handled are of 16 bit Size

2) Allows placing of code, data & Stack portions of the same program in different parts of memory for data & code protection.

Physical address calculation:

The generation of physical address requires information from two registers which are

1) segment address from Segment registers
2) offset address from pointers & index registers

Memory Segments:	Segment Registers (16 bit)	Offset address (registers) (16 bit)
Code Segment	CS	IP
Stack Segment	SS	SP
		BP
Data Segment	DS	SI
Extra Segment	ES	DI

As shown in the above table, the Segment registers and offset registers for different memory Segments are

for Code Segment:

This Segment is used to hold the program to be executed & the instructions are fetched from the Code Segment

CS register holds the 16 bit base address for this Segment. IP register (Instruction pointer) holds the 16 bit offset address

for Data Segment :

It holds the general data. SI also Stores Source operands during String operations. SI register holds 16 bit offset address and DS register holds the 16-bit base address

for Extra Segment :

It also holds the general data. Additionally, this segment is used as destination during String Operations. ES holds Segment/Base addr DI holds offset address.

for Stack Segment :

It holds stack memory which operates in LIFO Manner. SS holds its Base address. SP (stack pointer) holds the 16 bit offset at the top of the stack. BP (Base pointer) holds the 16 bit offset address for Random access.

The physical address has to be of 20 bits as the address bus is of 20 bits and the formula is given by.

Physical address = Segment address x 10 H + offset address

Eg if Segment address = 1234 H &
offset address = 0005 H then

Physical address = 1234H x 10H + 0005 H.
= 12340H + 0005H.
= 12345H.

**Note :
1) Physical address is also known as EFFECTIVE ADDRESS
2) SEGment address is also known as BASE ADDRESS

This formula ensures that the minimum Size of a Segment is 10 H Bytes.

Memory addresses and physical Memory organisation:

An 8086 has a 16-bit data bus, it should be able to access 16-bit data in one cycle. To do so it needs to read from 2 memory locations, as one memory location carries only one byte. The ~~16-by~~ 16-bit data is stored in two consecutive memory locations. However, if these two memory locations are in the same memory chip they cannot be accessed at the same time, as the address cannot generate two addresses simulatanously.

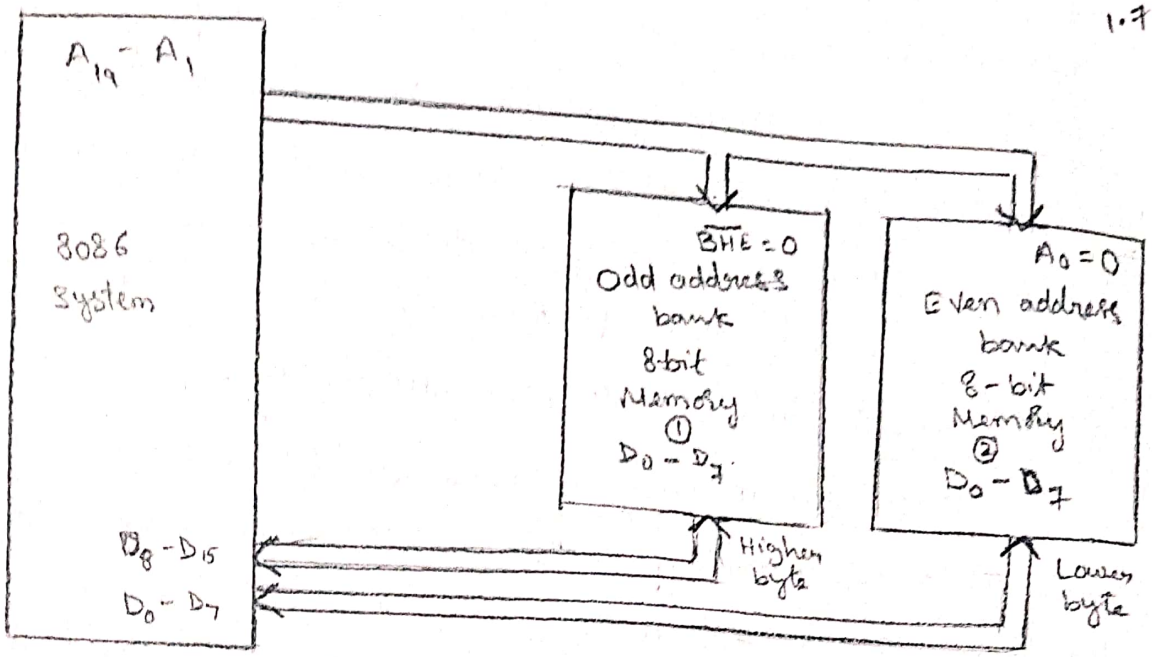
Hence the memory of 8086 is divided into two banks. Such that each bank provides 8 bits on a single address on the bus. The division is done such # a manner that any two consecutive locations lie in two different chips. Hence each chip contains alternate locations.

One Bank contains all even addresses called "even bank" while the other is called "odd Bank" containing all odd addresses Generally for any 16-bit operation, the even bank provides the lower byte and the odd bank provides the higher byte. The Even bank also called as lower bank and the odd bank is called the higher Bank.

The 1MB of 8086 μP is ~~diff~~ divided into two 512Kbytes of the of banks (even & odd). The Byte data with an even address is transfered on $D_7 - D_0$ while the byte data with an odd address is transfered 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 banks.

BIU (Bus Interface unit) requires two/one machine cycle



Based on the type of data stored in the even or odd address.

Misaligned data: If the word is stored at odd address then it is called Misaligned data and it requires ~~to~~ two Machine cycle for R/w (Read/write) operation to/from the memory.

Aligned data: If the word is Stored at even address then it is called Aligned data and it requires 1 machine cycle for R/w operation to/from the memory.

In order to maintain compatibility for the processor then the memory banking must be implemented in such a way that the even and odd banks must be selected for both 8 bit & 16 bit operations. This operation is summarised in the following table.

$\overline{BHE}$	A_0	operation
0	0	16 bit operation (R/w operation)
0	1	8 bit upper byte R/w operation
1	0	8 bit lower byte R/w operation
1	1	Idle state

→ The locations from FFFF0H to FFFFFH are reserved for operations like initialisation programme & I/O processor initialisa

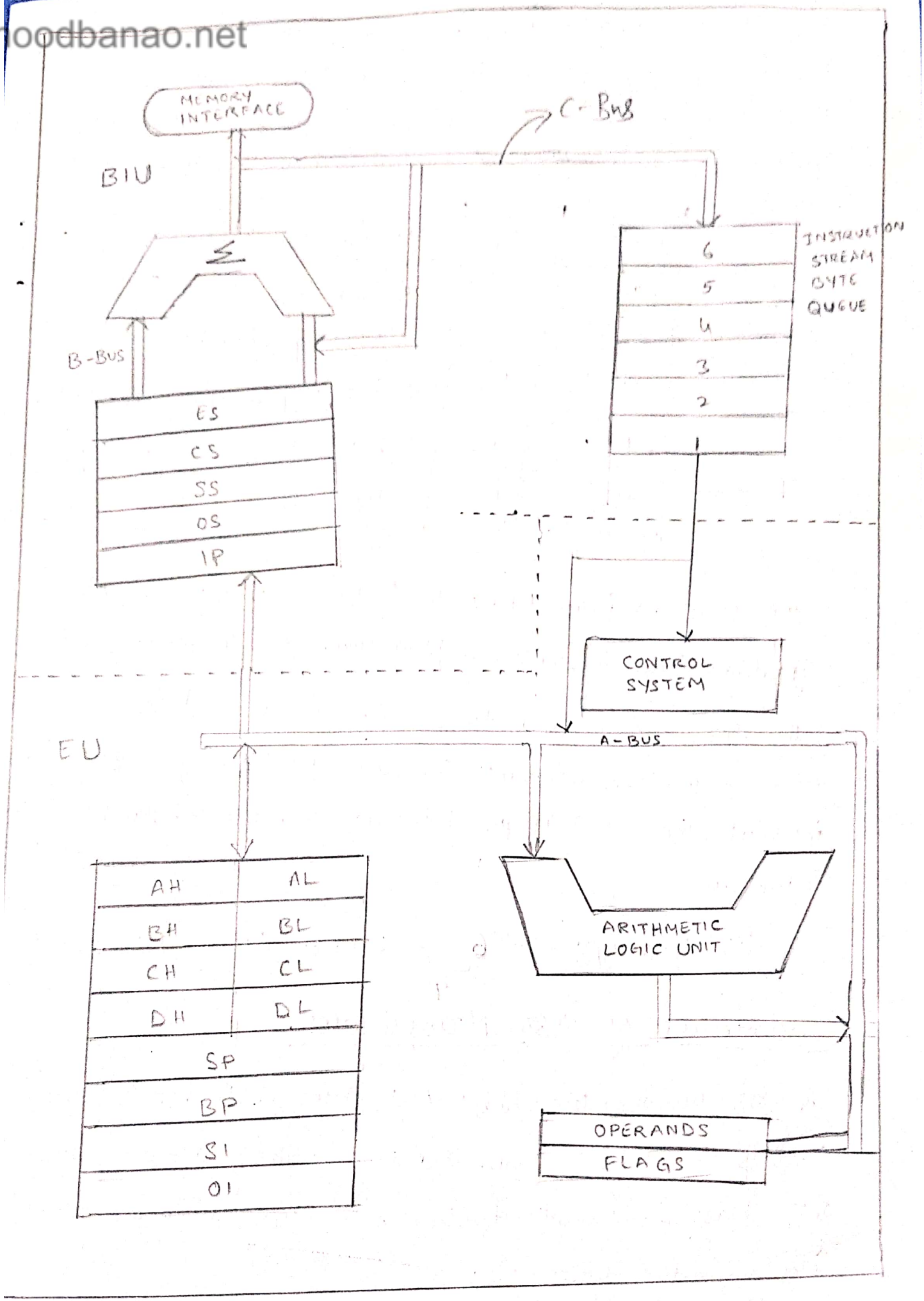
→ The locations 00000H to 003FFH are reserved for Interrupt Vector table. The interrupt structure provides space for 256 interrupt. The CS & IP for Interrupt require 4 byte for Interru Vector table

for 256 types $\cdot 256_{C_4}$ = 03FFH (1Kbyte).

Architecture of 8086 Microprocessor:

The 8086 μp supports a 16 bit ALU, a set of 16 bit registers and provides segmented memory addressing capability a rich instruction set, powerful interrupt structure, instruction queue for overlappe fetching and execution etc.

The internal block diagram is shown b as follows.



The complete architecture of 8086 can be divided into two parts.

(1) Bus Interface unit

(2) Execution unit.

Bus Interface unit:

The Bus Interface unit contains the circuit for physical Calculations and a 6-byte prefetch instruction queue. The bus interface unit makes the System's bus signals available for external interfacing of the devices. This means the unit is responsible for Establishing communications with external devices and peripheral including memory via the bus.

The 8086 addresses memory Segmentation. The physical address is 20 bit long and is generated by using Segment & offset registers.

physical address = Segment register H X 10H + Offset address.

The BIU implement the following functions.

1) Calculates the physical address
2) Fetches the instructions from memory
3) Supports instructions queue
4) reads / writes data to/from I/O devices and memory.

The other Components of the BIU are

1) Segment registers:

The Segment address value is to be taken from an appropriate Segment register depending on the whether the code, data

or Stack are to be accessed while the offset may be the Content of IP, BX, SI, DI, SP & BP or immediate 16 bit value depending on the addressing mode.

2) 6 Byte Pre-Fetch queue

It is a 6 Byte FIFO RAM used to implement pipeling. Fetching the next instruction while executing the current instruction is called pipelining.

The BIU fetches the next Six instructions - bytes from the Code Segment and stores it in the queue. The Execution unit (EU) removes instructions from the queue and executes them. The queue is refilled when atleast two bytes are Empty as 8086 has a 16-bit data bus. pipelining increases the efficiency of µp. But it fails when a branch occurs, as the pre-fetched instructions are no longer useful and it is discarded from the queue.

Execution unit:

It fetches instructions from the queue in BIU, decodes & Executes them. It performs arithmetic, logic and internal data transfer operations.

It Sends request signals to the BIU to access the External module.

The main components of the Execution unit is the ALU, Register bank and Timing & control circuit. It has all the registers in EU Except Segment registers & IP.

It has a 16 bit ALU able to perform arithmetic and logic operations. The 16 bit flag register reflects the results of execution by the ALU.

The Timing and control circuit derives the necessary control signals to execute the instruction opcode recieved from the queue; depending on the information available by the decoding circuit. The execution unit may pass the results to the bus interface unit for storing them in the Memory.

Signal description / Pin Configuration of 8086:

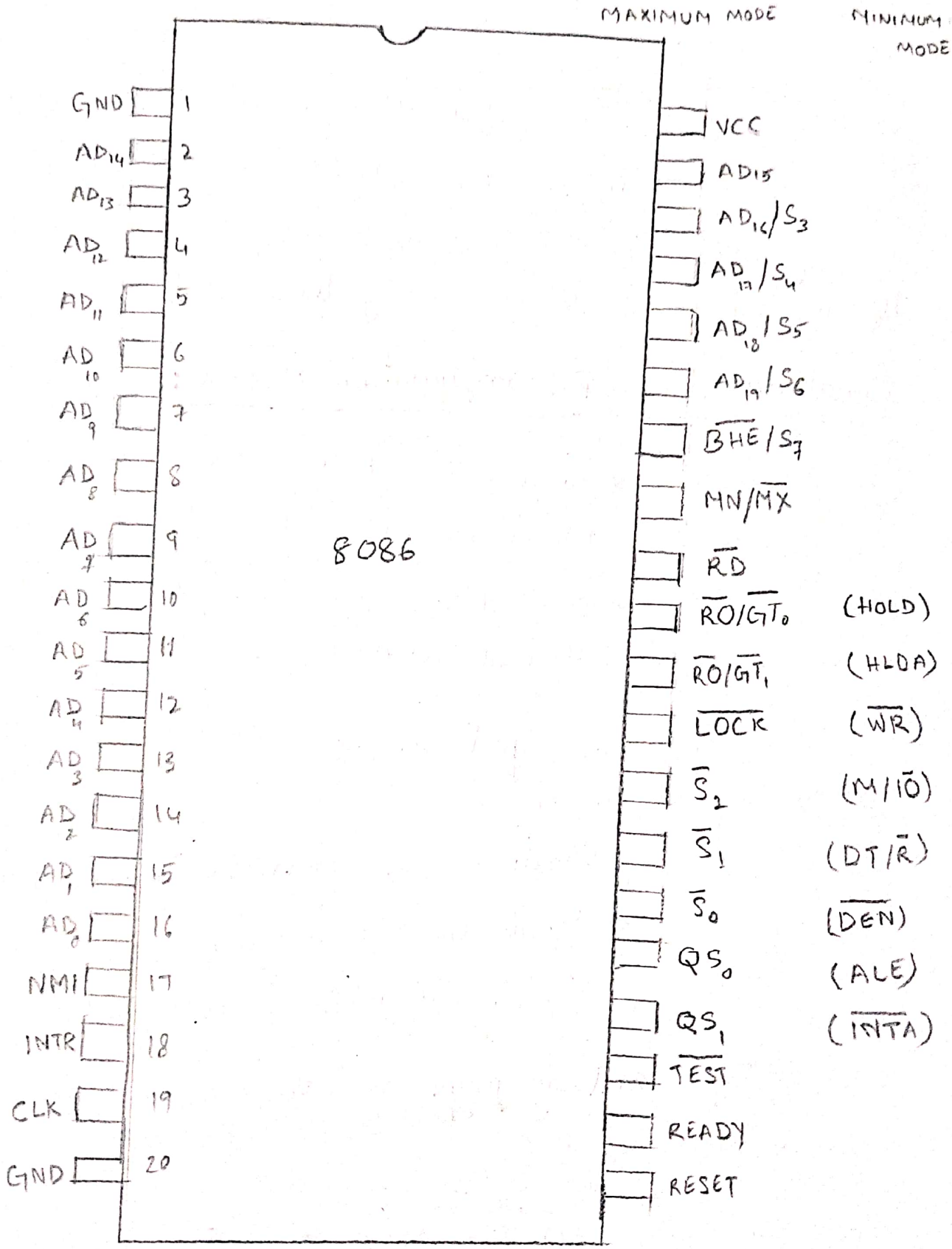
The Microprocessor 8086 is a 16-bit CPU available in three clock rates 5, 8, 10 MHz packaged in a 40 Pin CERDIP or Plastic package. The 8086 operates in single processor or multiprocessor Configuration to achieve high performance.

The 8086 signals can be classified into three groups. The first are the

1) Signals having common functions in both Max & Min modes.
2) Signals in maximum only
3) Signals in minimum only.

The maximum and minimum are the multiprocessor & single processor respectively.

The Signal descriptions are common for both minimum & maximum mode.



~~AD_{15} - AD_0~~ AD_{15} - AD_0: These are the time multiplexed address and data lines. Address remains on the bus during T_1 state and data is available on T_2, T_3, T_w & T_4 states. Hence T_1 T_2 T_3 & T_4 are clock states of machine cycles. T_w is wait states.

A_{19}/S_6, A_{18}/S_5, A_{17}/S_4 A_{16}/S_3: These are address and status Multiplexed on the same line. when clock is at T_1 the address is released and during the other state status will be released. (T_2 T_3 T_4)

S_6 - It indicates whether 8086 is Bus master or not (logically 0)

S_5 - It sends the status of interrupts flag, if $S_5 = 0$; IF = 0 else IF = 1

S_3, S_4 - Indicate which segment register is used. The combinations are as follows

S_4	S_3	Registers used
0	0	Extra
0	1	Stack
1	0	Code (or) none
1	1	Data

$\overline{BHE}/S_7$: BHE Stands for Bus High Enable. it is used to select the odd/higher address memory Bank to transfer the data over higher order (D_{15}-D_8). It is low during T_1 for read, write etc The Status information S_7 is available during T_2, T_3 T_4 and is not Currently used.

$\overline{RD}$: Read Signal when low indicates that processor needs to perform a memory (or) I/o operation. It is active low during T_2, T_3, T_4 or any read cycle.

INTR: Interrupt request is a level triggered input that is sampled during last clock cycle of each instruction to determine the availability of the request. It can be masked internally by resetting the interrupt enable flag.

Test: It is examined by wait. If the Test input goes low, execution will continue else the processor remains in idle state.

NMI: Non maskable interrupt is an edge-triggered input which causes Type 2 interrupt. It cannot be masked by software internally.

RESET: The input causes the processor to terminate the current activity and start execution from FFFF0H. This must be active high and must be active for atleast four clock cycles.

CLK- clock input provides the basic timing for processor operation and bus related activity. It is an asymmetric square wave with 33% duty cycle. The operating frequency of 8086 is 5MHz to 10MHz.

Vcc: +5V power supply for internal operation of circuit

GND: ground for internal circuit.

MN/$\overline{MX}$: The logic level at this pin decides whether the processor is to operate in either minimum or maximum.

Pins in Minimum mode

M/$\overline{I/O}$: If Memory / Input output operation is low then it indicates I/O operation, when this pin is high it indicates the CPU will have to perform memory operation. This line will be active till the T_4 clock cycle.

$\overline{INTA}$: Interrupt acknowledge is a read strobe for interrupt acknowledge cycles. It means that the processor has accepted the interrupt it is active low during T_2 T_3 & T_W of each acknowledge cycle.

ALE: Address Latch Enable: This output signal indicates the the availability of address on the address/data lines, and is connected to latch enable input of latches. This signal is active high and is never tristated.

DT/$\overline{R}$ - Data transmit/Receive: This output is used to decide the direction of data flow through the transreceivers. when the Processor transmits the data the signal is high & if the processor receives the data this is '0' (low)

$\overline{DEN}$ - Data Enable: The signal indicates the availability of valid data over the address/data lines. It is used to enable the transreceivers to seperate the data from the multiplexed address/data signal. It is active from the middle of T_2 and Middle of T_4.

Hold, HLDA - Hold/Hold Acknowledge: when the HOLD line goes high, it indicates to the processor that another master is requesting the bus access. The processor, while after receiving the HOLD request issues Hold acknowledgment on HLDA pin, after the current instruction cycle.

These pins are mainly used by DMA (Direct Memory Access). If DMA requests the bus while the CPU is performing Memory or I/O cycle. then it will release the bus after T_4 provided:

1) request occurs on or before T_2 of the current cycle.
2) A lock instruction is not being executed.

Pins in Maximum Mode:

$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$ - Status lines: These are the status lines which indicate the type of operation carried out by the processor. These becomes active during T_4 of the previous cycle and remain active during T_1 & T_2 of the current bus cycle.

These status lines are generated by (encoded as a by Bus Controller as shown below

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	operation	Signals generated by 8288
0	0	0	Interrupt acknowledge	$\overline{INTA}$
0	0	1	Read I/O port	$\overline{IORC}$
0	1	0	Write I/O port	$\overline{IOWC}$
0	1	1	Halt.	None
1	0	0	Code Segment / Instruction fetch.	$\overline{MRDC}$
1	0	1	Read memory	$\overline{MRDC}$
1	1	0	Write memory.	$\overline{MWTC}$
1	1	1	Inactive.	None

$\overline{LOCK}$: This output pin indicates that the other system bus masters will be prevented from gaining the system bus, while the $\overline{LOCK}$ signal is low. The $\overline{LOCK}$ signal is activated by the "lock" prefix instructions and remains active until the completion of the next instruction.

QS_1, QS_0 - Queue Status: These lines give information about the status of the code-prefetch Queue. These are active during the CLK cycle after which the queue operation is performed. These are encoded as below

QS_1	QS_0	operation
0	0	No operation
0	1	First byte of opcode from queue.
1	0	Empty queue.
1	1	Subsequent byte from the queue

$\overline{RQ}/\overline{GT_0}$, $\overline{RQ}/\overline{GT_1}$ - Request/Grant: These pins are used by other local bus masters, in maximum mode, to force the processor to release

the local bus at the end of processor's current bus cycle. All pins are Bidirectional and the pin $\overline{RQ}/\overline{GT_0}$ has higher priority than $RQ/\overline{GT_1}$,

Queue & its need in 8086 8086 µp:

This queue operation is based on FIFO (first in first out) concept, such that the Execution unit gets the instruction for Execution in the order they are fetched.

Thus it is also called as opcode fetch FIFO Buffer.

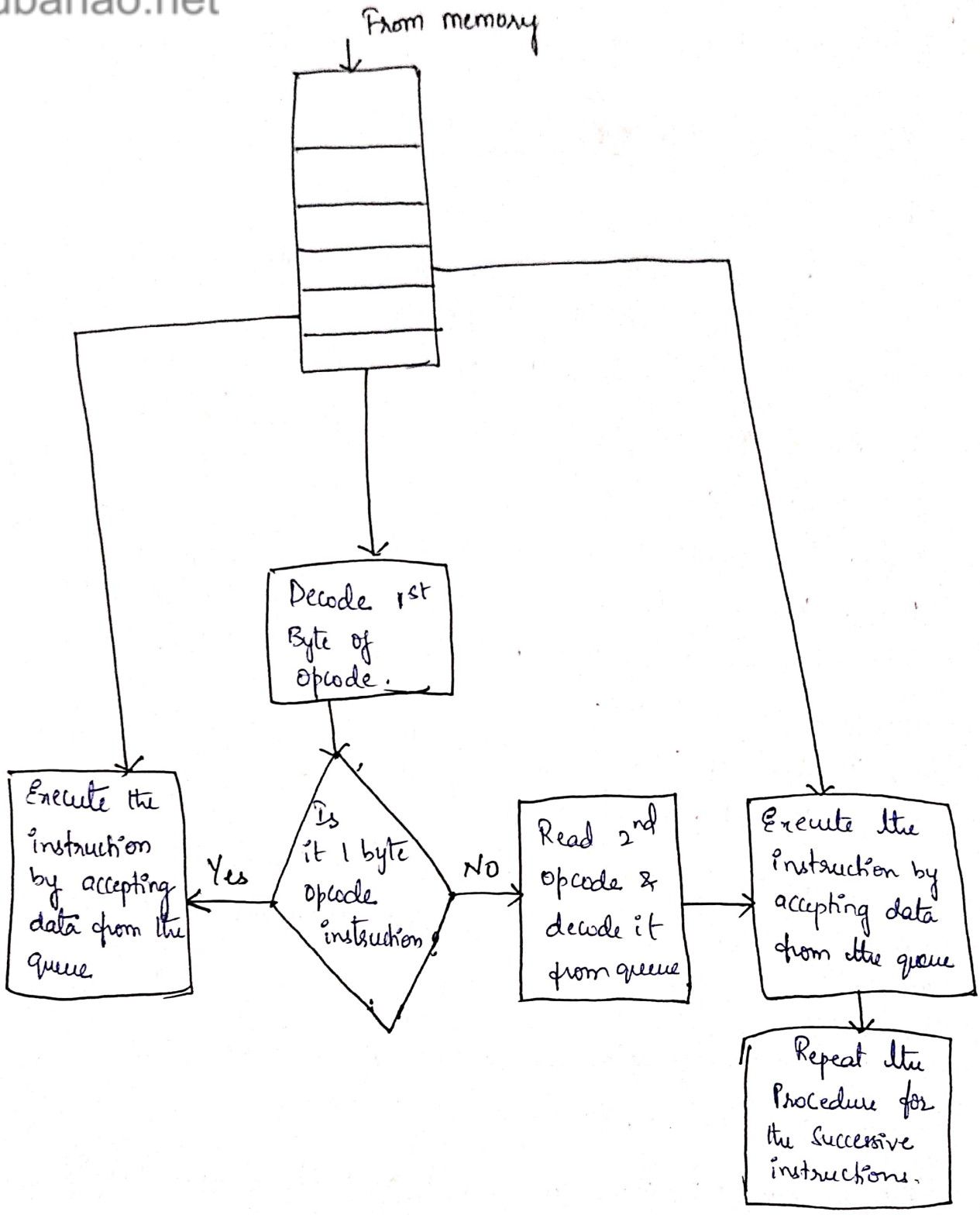
To speed up program Execution, the BIU function fetches six bytes of instructions ahead of time from the memory & stores it in FIFO Buffer. These instructions are stored in a group of registers called queue. when the current instruction is being Executed, the next instruction will be fetched and the process continues as long as the queue is not full.

In case of JUMP & CALL, the instructions which are already fetched are of no use and the queue is flushed out.

This process of fetching the next instruction while the current instruction is being Executed is called "Instruction Pipeling"

The queue operation is explained in the figure below as or the process is as follows.

1) initially the instructions are fetched and stored in 6 byte queue.
2) Decode the 1st byte of the opcode. The decoder circuit checks if the instruction is of ~~two~~ one byte or more than that.
3) If the length of the instruction is one byte then the instruction is taken and executed along with the data.
4) If the length is more than one byte then the next byte will be ~~addr~~ fetched and the entire execution is performed.



The queue will be updated only after ~~two~~ every bytes are is read from the queue but the fetch cycle is initiated by BIU only if at least two bytes of the queue are empty.

8086 Interrupts:

An Interrupt breaks the normal sequence of Execution of instructions, diverts its execution to some other program called "Interrupt Service routine". After Executing ISR, the Control is transfered back again to the main program

The Interrupts can broadly be classified as 1) Hardware Interrupts
2) Software interrupts.

The Hardware interrupts are called as External interrupts. 8086 µp has two pins to accept hardware interrupts they are : NMI & INTR.

The Software interrupts are called as Internal interrupts. 8086 has 256 Software interrupts. These can be invoked by the programmer by using INT n where n is the number of interrupt n = 0 to 255.

Another Source of interrupt can be generated while executing certain program

Eg An divide by Zero interrupt which automatically invokes INT 0

Process of Services to the interrupts:

Whenever an interrupt occurs then the processor follows the Procedure

Response to any interrupt - INT N:

Step 1: The µp will push Flag register into stack

Step 2: Clear IF & TF flag register and thus disables INTR int

Step 3: push CS to the stack

Step 4: push Ip into the stack

Step 5: load New Ip & New CS from the IVT

Every Interrupt Service routine ends with IRET. When IRET is encountered then the following procedure is followed.

Response to IRET

Step 1: Pop IP from the stack

Step 2: Pop CS from the stack.

Step 3: pop flag register from the stack.

Interrupt Vector table:

Interrupt vector table is a vector that stores the addresses of the 256 interrupts. Each ISR address is of 4 bytes (2 bytes for CS & 2 bytes for IP). As there are 256 interrupts 256×4 bytes = 1024 Bytes $\Rightarrow$ 1 KB. A total of 1KB is reserved for IVT. The first 1KB of memory address 00000 H to 003FFH are reserved.

Whenever an interrupt INT N occurs, MP calculates the ISR address As N×4 to get the values of IP & CS from IVT and hence perform the ISR.

The interrupts 0 to 4 are called as Dedicated interrupts.

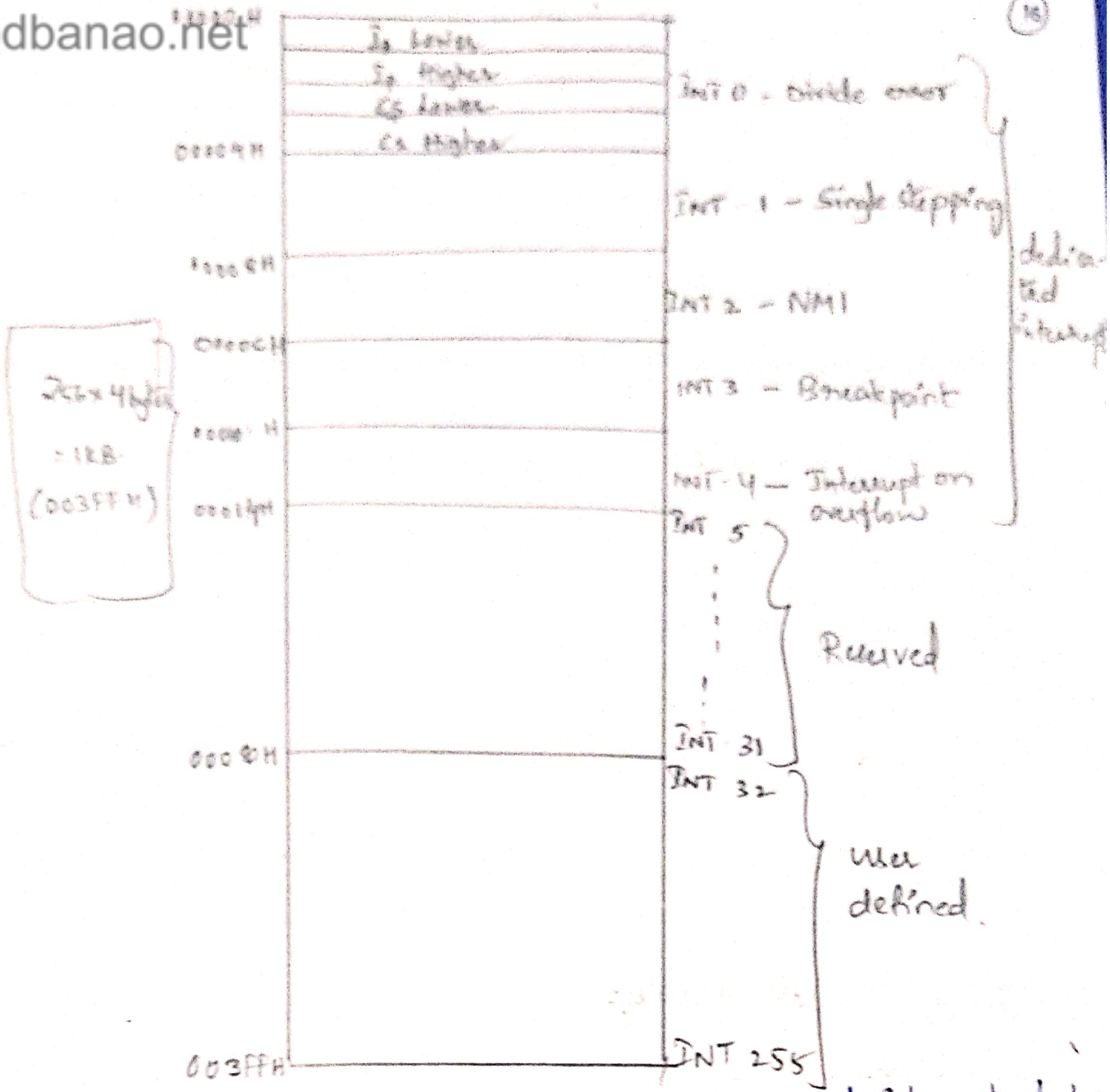
1) INT 0 - (Divide error) : It occurs when there is a division error.

2) INT 1 - (Single step) : It interrupts after every instruction if IF=1

3) INT 2 - (NMI) : It executes this ISR in response to an interrupt on NMI line.

4) INT 3 - (Breakpoint interrupt) : This interrupt is used to cause Breakpoints in the program. It is used in debugging large programs where Single stepping is inefficient.

5) INT 4 - (overflow interrupt) : The interrupt occurs if the overflow flag is set & the MP executes the INT 0 instruction. It is used to detect overflow error in signed arithmetic operations.



Reserved interrupts: INT 5 to INT 31 are reserved interrupts to be used in high end processors and are unavailable for the programmer

User defined interrupts: The interrupts INT 32 - INT 255 are called as user defined software interrupts whose ISR's are written by end user.

Hardware interrupts:

1. NMI (Non maskable interrupt): This is and edge triggered, high Priority interrupt. This interrupt when occurs, the MP executes INT - 2
2. INTR: It is Maskable, level triggered, low priority interrupt. When INTR occurs, the MP Executes two $\overline{INTA}$ pulses. At the first pulse the interrupting device prepares to send the vector number & at the second pulse the device sends the vector Number 'N' to the MP.

When two interrupts occur simultaneously then, the µp services the interrupt with the highest priority. The order of the priority of the interrupts from highest to lowest priority is

Divide by Zero, INT N, → NMI → INTR → Single stepping.

PART-2

INSTRUCTION SET & ASSEMBLY LANGUAGE PROGRAMMING of 8086 µp

Instruction formats:

A General Machine Language instruction format consists of two fields.

1) **OPCODE** : The opcode or operation code indicates the type of operation to be performed by the CPU.

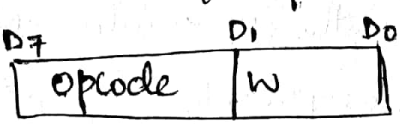
2) **OPERAND** : The operand defines the data on which the operation is to be performed.

There are 6 general formats of instructions in 8086 µp. The length of the instruction may vary from one byte to 6 byte

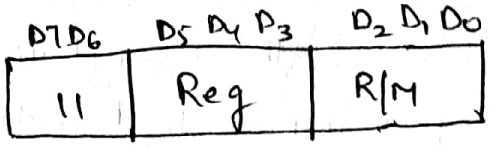
Types of instruction formates:

1. **One byte instruction** : It is only one byte long. & may have implied data or register operands. The least significant 3 bits specifies register operand, if any. If register is not present then all the 8 bit form the opcode.

2. **Register to Register** : It is 2 Byte long. First byte specifies opcode and width of operand is specified by W. The second byte shows the register and R/M field.



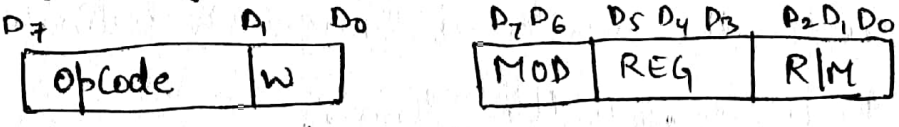
1st byte



2nd Byte

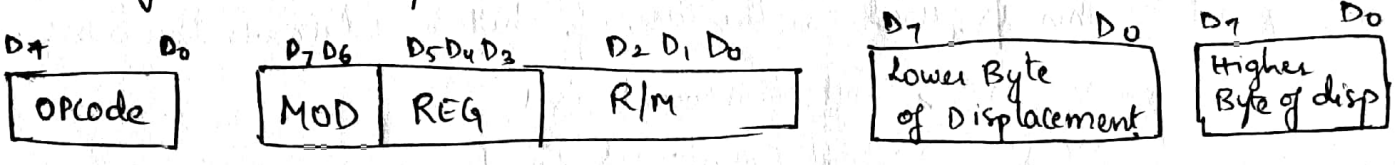
Where REG – Register (it is one operand)
R/M – represents register or memory location is another operand

3. Register to/from memory with no-displacement: It is 2 bytes long and is similar to register to register except MOD as shown

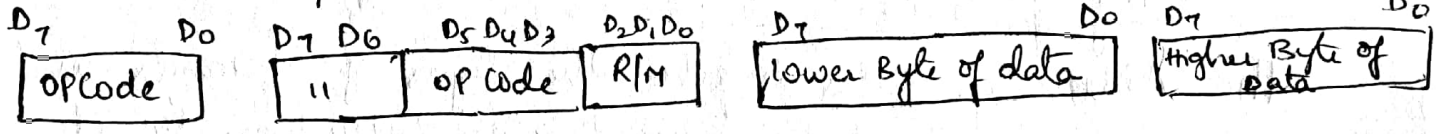


where MOD - represents Addressing mode
W - represents if the data is 8 bit or 16 bit.
if W = 0 → 8 bit
if W = 1 → 16 bit.

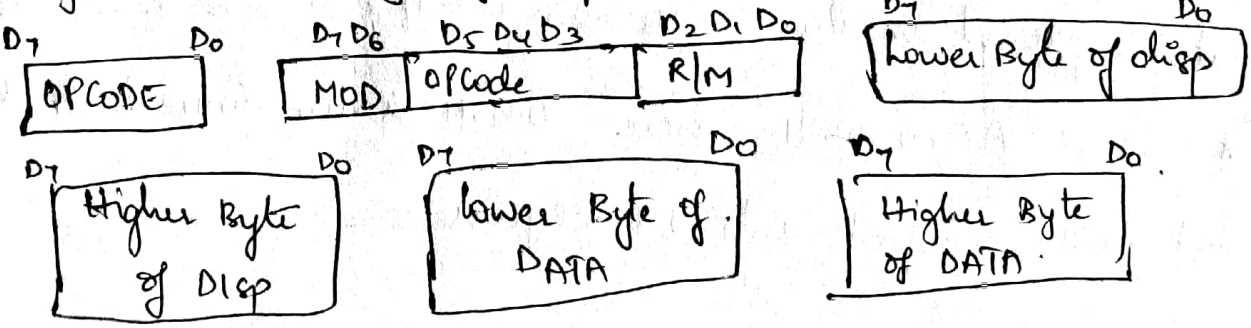
4) Register to/from Memory with displacement: This type of instruction format contains one or two additional Bytes for displacement along with 2 byte used as in the format of register to/from memory without displacement. The format is as shown below.



5 Immediate operand to Register: The first Byte as well as the 3 bits from second byte used by REG field in the case of register to register format are used for opcode. It contains one or 2 bytes of immediate data.



6. Immediate operand to memory with 16 Bit displacement:
It requires 5 or 6 bytes of coding. The first 2 bytes contain the information regarding OPCode, MOD and R/M fields. The remaining 4 bytes contain 2 bytes of displacement & 2 bytes of data as shown below



The opcode have single bit indicators. Such as

1. W-bit → decides if the instruction is 8bit/16bit.
2. D-bit → decides if the register is source or destination

when d=0 register is the source.
d=1 register is the destination

S-bit: This bit is called sign extension bit. The S-bit along with W bit to show the type of operation.

S	W	operation
0	0	8 bit operation with 8-bit immediate operand
0	1	16 bit operation with 16 bit immediate operand.
1	0	16 bit operation with a sign extended immediate operand

V-bit: This is used in the case of shift & rotate instructions.

If V=0 → shift count is 1
V=1 → shift count is in CL

Z bit: The z-bit is used in REP instruction if z=1 then the instruction with REP executes until zero flag is (1).

Addressing Modes of 8086 µp:

The Addressing Modes describe the types of operands and the way they are accessed for executing an instruction. i.e it indicate the way of locating data or operands.

Types of addressing modes:

1. Immediate addressing mode: In this mode, the immediate data is present in the instruction itself. The data can be 8/16 bits

Eg MOV AX, 0005H here 0005H is the immediate data that will be copied into AX register.

2. Register addressing Modes:

In Register addressing Mode, the data is stored in register & the register is passed as an operand. All registers except IP can be used.

Eg: MOV AX, BX ; //

In this the data is stored in BX and BX is passed as operand. The value in BX register is copied into AX register.

3. Direct addressing Mode:

In this mode, a 16-bit Memory address (offset) is directly specified in the instruction.

Eg: MOV AX, [5000 H]

Here data is in data segment whose segment address is given by DS register and 5000H will be the offset address.

Physical address = 10H * DS + 5000 H.

4. Register Indirect addressing Mode:

In this mode, the address of the data or operand is determined (or) stored using offset registers such as BX, SI or DI registers. The default segment is either DS or ES. As the address is not provided directly in the instruction it is known as Register indirect mode.

Eg: MOV BX, 2000
MOV AX, [BX]

Here Data is present in location 2000 of DS. The 2000 (offset address) is stored in register BX.

physical address = 10H * DS + [BX] here BX = 2000

5. Indexed Addressing Modes:

It is a special case of indirect addressing Mode. The offset address is stored in one of the index registers (SI or DI). DS is the default segment for index registers SI & DI.

Eg: 1) MOV AX, [SI] (or) 2) MOV AX, [DI]

As mentioned the offset address will be stored in SI and the data at the address carried by SI will be copied to AX.

Physical address = 10H * DS + [SI]

6. Register Relative Addressing mode:

In this mode, the data is available at an effective address formed by adding an 8 bit or 16 bit displacement with the content of any one of the registers BX, BP, SI or DI. When BX, DI, SI are used Default segment is DS or ES when BP is used the default segment will be stack segment.

Eg: 1) MOV AX, 50H[BX]

The data to be copied into AX is present at the location formed by.

address 10H * DS + 50H + [BX]

2) MOV CX, 20H[BP]

The data to be copied into CX register is at the location pointed by address 10H * SS + 20H + [BP] as the BP register is used default segment is stack.

7. Based Indexed:

The effective address of data is formed, by adding contents of Base register (BX or BP) to the content of an index register in the default segment.

Eg: 1) MOV AX, [BX][SI]

The data to be copied into AX register is at the location pointed by address

10H * DS + BX + SI.

8. Relative Based indexed:

The Effective address is formed by adding an 8 or 16 bit displacement with the sum of contents of any one of the Base registers (BX or BP) and any one of the index registers, in a default segment.

Eg: 1) MOV AX, 50H[BX][SI]

The data to be copied into AX register is at the location pointed by address

10H * DS + BX + SI + 50.

2) MOV CX, 20H [BP][SI]

The data to be copied into CX is located at the address pointed by

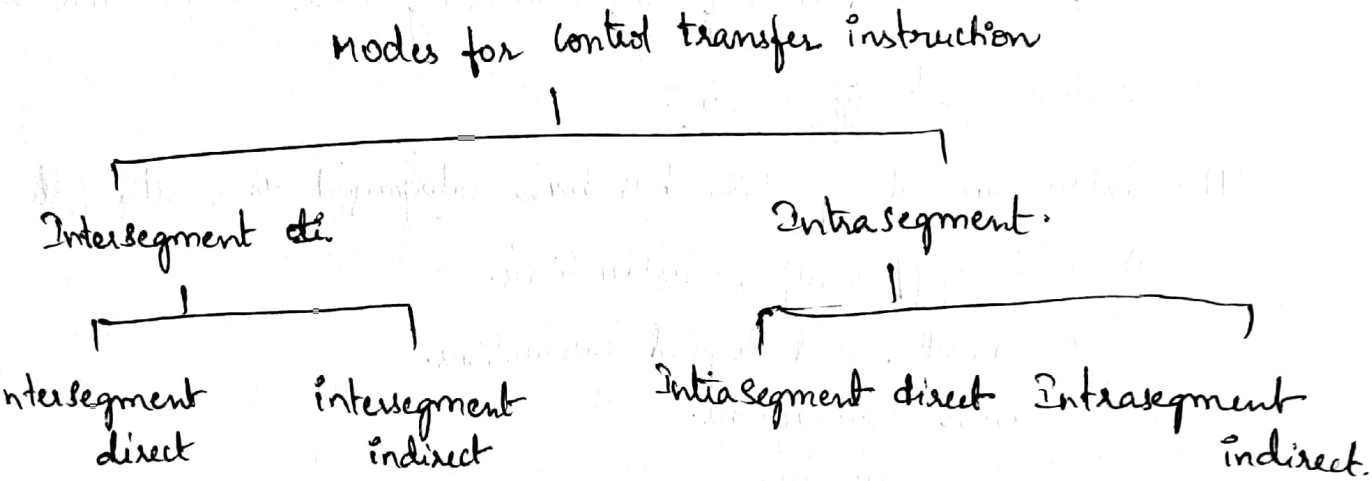
10H * SS + BP + SI + 20H [Here default segment is stack segment as BP is used]

For control transfer instructions:

The addressing modes depend upon whether the destination location is within the same segment or in a different one.

If the location to which the control is to be transferred to a different segment, it is called intersegment mode

If the control is in the same segment it is called intrasegment mode. The addressing modes can be classified as follows.



9. Intrasegment Direct Mode:

In this mode, the control to be transferred will be in the same segment and will be directly given in an instruction as a immediate displacement

The displacement is computed relative to Instruction pointer. Eg: JMP the displacement can be 8 bit or 16 bit. If the displacement is 8 bit it is short jump in range $(-128 < d < +127)$ & if 16 bit it is long jump in range $-32768 < d < +32767$

10. Intrasegment Indirect Mode:

In this Mode also, the control to be transferred will be in the same segment, but the displacement is passed to the instruction indirectly

11. Intersegment Direct:

In this mode, the control to be transferred will be in different segment. It provides a means of branching from one code segment to another CS. Here CS & IP of the destination are specified directly in the instruction

Eg: JMP 5000H : 2000H;

here 5000 is the CS address & 2000 H is offset address.

12. Intersegment Indirect:

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

Instruction Set of 8086:

The instruction set of 8086 has been categorized into the following types

1) Data copy/Transfer instructions.
2) Arithmetic and logical Instructions.
3) Branch Instructions.
4) Loop Instructions.
5) Machine control Instructions.
6) Flag Manipulation Instructions.
7) Shift & rotate instructions.
8) String Instructions.

Data Copy | Transfer Instructions: i) MOV :- Move

These instructions transfer data from one register | memory location to another register | memory location.

The source may be
1) Segment registers
2) general purpose register
3) special purpose register
4) Memory location

The destination can be 1) Register
2) Memory destination.

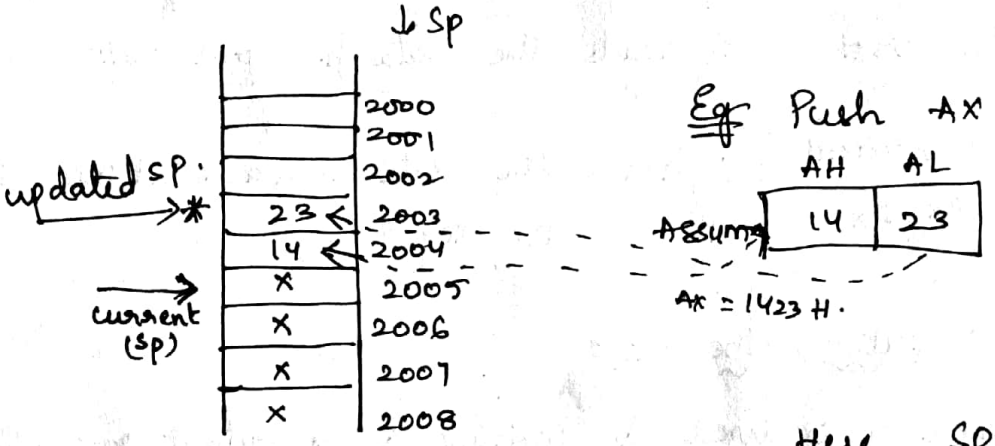
Note: In immediate addressing mode, a Segment register cannot be a destination.

Eg: 1) MOV AX, 5000H → valid ; Immediate
2) MOV AX, BX → valid ; Register
3) MOV AX, [SI] → valid ; Indirect
4) MOV AX, [2000H] → valid ; direct
5) MOV DS, 5000H → invalid ; as the Segment reg can't be loaded directly
6) MOV AX, 5000H → In order to load the value in DS, it has to be moved into GPR first & then moved it into DS
MOV DS, AX

3. Push: Push to Stack:

This instruction pushes the contents of the specified Register / Memory location on to the Stack. The Stack pointer is decremented by 2, after each Execution. The current Stack-top is always occupied by the Previously pushed data.

Hence the push operation decrements the SP by two and Then Stores the two byte contents of the operand onto the Stack. The Operation is shown below.

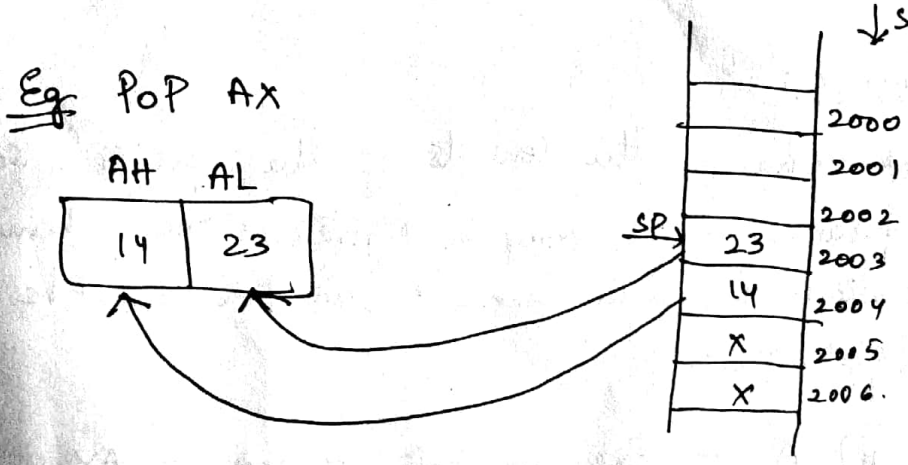


Here $Sp = Sp - 2$

Part of memory (SS)

4. POP: pop from Stack:

This instruction when loaded (Executed), loads the specified Register / Memory location with the contents of the memory location of which the address is formed using current Stack Segment & Stack Pointer as usual. The Stack pointer is incremented by 2.



Here $sp = sp + 2$

5 IN: Input to the port:

This instruction is used for reading an input port. The address of the input port may be specified in the instruction directly or directly. AL and AX are the allowed destinations for 8 and 16 bit input operations. Dx is the only register which is used to carry 16 bit address. If the port address is 16 bit. It must be definitely in Dx register

Examples:

1. IN AL, 03H ; reads the data from port 03H.

2. MOV DX, 0800H
   IN AX, DX
   } ; reads the data from a 16 bit port address passed through Dx register.

6 OUT: Output to the port:

This instruction is used for writing to an output port. The address of the output port may be specified directly (8 bit) specified in the instruction or can be through Dx register. (16 bit). The registers AL & AX are allowed source operands for 8 bit and 16 bit operations respectively.

Examples:

1) OUT 03H, AL ; Sends data to port 03H.

2) MOV DX, 0300H
   OUT DX, AX
   } ; sends data to port 0300H, to

7) XCHG: Exchange:

This instructions Exchanges the contents of the specified source and destination operands, which may be registers or one of them may be a memory location. But both source & Destination cannot be Memory location at once.

Eg XCHG [5000H], AX ; Exchanges contents of 5000H & AX

8) XLAT: Translate:

This instruction is used for finding out the codes in case of code conversion problems, using look up table technique.

Eg When a key is pressed in a Keypad, the corresponding binary code is sent to 8086 so that the processor understands which key is Pressed. once a key is pressed and need to be displayed on display device (7 seg display). 7-seg display also works on binary code.

This translation from the code of the key pressed to the corresponding 7-segment code is performed using XLAT Translation.

9 LEA: Load Effective address:

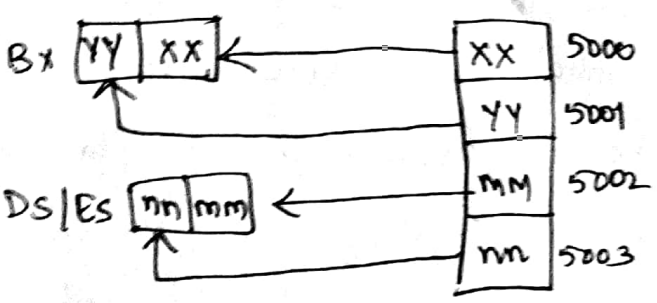
This instruction loads the effective address formed by destination operand into the specified source register. This instruction is more useful for assembly language rather than for Machine language.

Eg LEA BX, ADR ; loads the address of the label ADR.

10 LDS/LES: load pointer to DS/ES:

This instruction loads the DS or ES register and the specified destination register in the ~~register~~ instruction with the content of memory location specified as a source in the instructions.

Eg LDS BX, 500H / LES BX, 500H



11) LAHF : Load AH from lower Byte of flag :

This instruction loads the AH register with the lower Byte of the flag register. This command may be used to observe the status of all the conditional flags except (overflow) at a time. (AH ← Flag register)

12) SAHF : Store AH to lower Byte of flag register.

This instruction sets or resets the conditional flags in the lower Byte of the flag register depending upon the corresponding ~~to the~~ bit position in AH. (AH → Flag register)

Eg If AH = 0000 0000 -All the bits in the lower Byte of the flag register are reset.

13 PUSHF : Push flags to stack :

The push flags instruction pushes the flag register on to the stack & the sp is decremented by 2. It is used in the case interrupts.

14 POPF : Pop flags to stack :

The pop flags instruction ~~pushes~~ Pops the flag register from the stack & the sp is incremented by 2.

Arithmetic instructions :

1. ADD : Add

This instructions adds the contents of source and destination and stores the result in destination register (or memory location).

→ The Source can be Register / Memory location / Immediate Value.
→ The destination can be Register / memory location
→ The source and destination cannot be Memory location at once
→ Segment registers cannot be added
→ All the conditional flags can be affected depending on result

Eg: 1) ADD AX, 0100H → AX = AX + 0100H.

2) ADD AX, BX → AX = AX + BX

3) ADD AX, [SI] → AX = AX + [SI]

4) ADD AX, [5000H] → AX = AX + [5000H]

5) ADD [5000H], 0100H → [5000H] = [5000H] + 0100 H.

2. ADC: Add with Carry:

This instruction performs the same operation as ADD instruction, but adds the Carry flag bit (which may be set as a result of the previous Calculations) to the result. All the Conditional flags are affected based on result.

Eg: 1) ADC AX, BX → AX = AX + BX + CY

2) ADC AX, [SI] → AX = AX + [SI] + CY

3) ADC AX, [5000H] → AX = AX + [5000H] + CY.

3 SUB: Subtract

The Subtract instruction subtracts the source operand from the destination operand and stores the result in destination operand.

→ The Source can be an immediate data / Register / memory location

→ The destination can be a register / memory location

→ It Cannot hold Memory locations in both source & destinations at a time

→ All Conditional flags are affected based on the result.

Eg: SUB AX, 0100H → AX = AX - 0100H.

SUB AX, BX → AX = AX - BX

SUB AX, [5000H] → AX = AX - [5000H]

4 SBB: Subtract with Borrow:

This instruction subtracts source operand, Carry flag from the

destination operand. The result is stored in destination operand. All the conditional flags are affected.

Eg: SBB AX, 0100H → AX = AX - 0100H - CY

SBB AX, BX → AX = AX - BX - CY.

5) MUL: Unsigned Multiplication Byte or word:

This instruction multiplies an unsigned Byte or word by the contents of AL (AX). The data can be in one of the general purpose registers or memory locations.

→ If the result is 16 bits it is shown in AX register and if the result is 32 bits it is stored in DX, AX. DX contains Most significant word and AX contains least significant word.

→ All the flags are Modified based on the result.

→ Immediate data is not allowed.

Eg: i) MUL BH → (AX) = (AL) X (BH)

ii) MUL CX → DX, AX = AX X CX

6) IMUL: Signed Multiplication:

This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX.

Eg: IMUL BH

IMUL CX

7) DIV: Di unsigned Division:

This instruction performs unsigned division. It divides an

unsigned word or double word by a 8 bit or 16 bit operand respectively.

16 bit operation:

Dividend must be in AX & Divisor can be in any of the modes except immediate Addressing mode.

The remainder & the quotient will be in AH & AL respectively.

32 bit operation:

Dividend must be in DX (MSW) & AX (LSW) and the divisor can be in any of the modes except immediate Addressing mode.

The remainder & the quotient will be in DX & AL respectively

Eg: DIV BL ; AX = AX/BL

8 IDIV: Signed Division:

This instructions perform same operation as the DIV instruction, but with signed operands. All the results are stored similar to DIV instruction. The result will be signed numbers only.

All the flags are undefined after IDIV instructions.

9 and INC: increment

This instruction increases the content of the specified register or memory location by 1. All the conditional flags are affected Except Carry.

Immediate data cannot be an operand of this instruction.

Eg: INC AX ⇒ AX = AX + 1

10 DEC: Decrement:

This instruction decreases the content the specified register or

Memory location by 1. All conditional flags are affected Except Carry. Immediate operand cannot be used.

Eg DEC AX ⇒ AX = AX - 1

11 CMP : Compare :

This instruction compares the source operand with the destination operand. The source can be immediate / Register / Memory location & the destination can be register / Memory location.

for comparision, it subtracts the source operand from the destination operand by does not store the result anywhere. The flags are affected depending on the result of Subtraction.

If carry flag and Zero flag are both '0' it means, A = B.

If source operand is greater than destination operand, Carry flag is Set else Carry flag is reset.

Eg 1) CMP BX, 0100H → BX - 0100H.

2) CMP AX, BX → AX - BX

12 NEG : Negate :

The negate instruction form 2's complement of the specified destination in the instruction. For obtaining 2's complement, it subtracts the contents of destination from Zero. & the result is stored back in the destination operand which can be a register / memory location.

Eg A. NEG AL ⇒ AL = 0000 - AL

13 CBW: Convert Signed Byte to word:

This instruction converts a Signed byte to a Signed word. in other words, it copies the sign bit of a byte to be converted to all the bits of the higher byte of the result word. The Byte to be Converted must be in AL, The result will be in AX.

Eg: 1) MOV AL, 22h.

CBW ⟶ AX = 0022H.

2) MOV AL, F0H.

CBW ⟶ AX = FFF0H.

14 CWD: Convert Signed word to Double word:

This instruction Copies the sign bit of AX to all the bits of the DX register. This operation is to be done before Signed division. It does not affect any flag.

15 DAA: Decimal Adjust Accumulator:

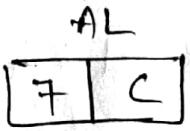
This instruction is used to convert the result of the addition of two Packed BCD numbers to a valid BCD Number. The result has to be only in AL

1) If Lower nibble is greater than 9 (or) if AF is 1 then add 06 to the lower nibble of AL.

2) After adding 06 in the lower nibble of AL, if the upper nibble of AL is greater than 9 or if the Carry flag is set, add 60H to AL.

Eg ① Consider AL = 53 ; CL = 29

AL + CL = 53
+29
AL = 7CH



As C > 9, 06 is added to 7C

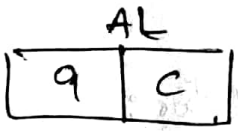
⇒ 1
7C
+06
AL = 82

Instructions :
```
MOV CL,29
MOV AL,53
Add AL, CL
DAA
INT 3
```

Example 2 :

Assume AL = 73 & CL = 29 AL + CL = 73 + 29

Step-1) 73
29
9C



As C > 9 06 must be added to 9C

Step-2 →) ∴ 1
9C
+06
→ A2

Here as A > 9 60 must be added to A2

∴ Step-3 A2
60
→ 1 02
Carry

∴ The result 02 will be stored in AL & the Carry flag is set.

16 This instruction converts the result of subtraction of two Packed BCD numbers to a valid BCD Number. The subtraction has to be in AL only.

If the lower nibble of AL is greater than 9, the instruction will subtract 06 from lower nibble of AL.

If the result of subtraction sets the carry flag or if upper nibble is greater than 9 it subtracts 60H from AL. It modifies. AF, CF, SF, PF & ZF flags.

Example: ① if AL = 75 & BH = 46

Step 1: AL - BL ⇒

$$\begin{array}{r} {}^{6}{}^{16} \\ 7\cancel{5} \\ -\ 46 \\ \hline 2F \end{array}$$

here AF = 1 & as F > 9 06 must be subtracted

Step-2:

$$\begin{array}{r} 2F \\ -\ 06 \\ \hline 29 \end{array}$$

AL = 29

Example-2:

If AL = 38 & CH = 61

1) AL - CH =

$$\begin{array}{r} {}^{16} \\ +\cancel{3}8 \\ -\ 61 \\ \hline D7 \end{array}$$

As D is > 9 & CF = 1 (borrow) D - 6 Must be performed

2)

$$\begin{array}{r} D7 \\ -\ 60 \\ \hline 77 \end{array}$$

AL = 77

17 AAA: ASCII Adjust After Addition:

The AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. The AAA instruction converts the resulting contents of AL to unpacked decimal digits. After addition, the instruction examines.

① If lower nibble of AL is a number in 0 to 9 & AF is Zero (0) then AAA sets the higher Nibble to 0.

② If lower nibble of AL is between 0 to 9 & AF is One (1) then 06 is added to AL
AH is made 00 & then AH is incremented by 01

③ If the lower nibble of AL is greater than 9 &
then AL = AL + 06
AH = 00
AH = 01
AF = 1
CF = 1

In this instruction other flags are unaffected.

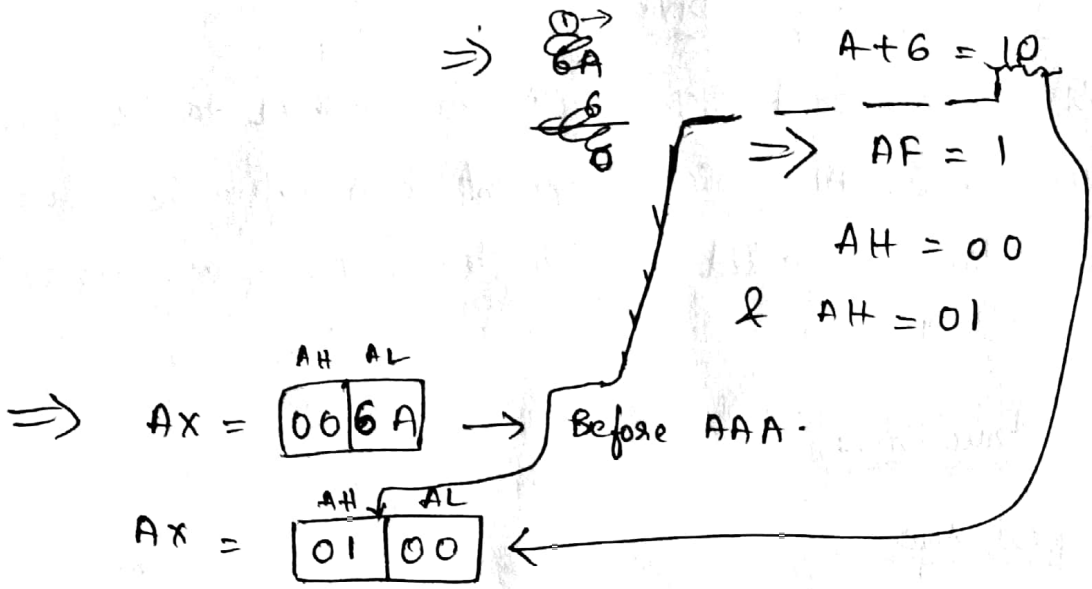
Eg 1: Let AL = 67.

As lower Nibble is in the range of 0 to 9 & AF is undefined then the higher Nibble = 0

⇒ AL = 67 Before AAA.
AL = 07 After AAA

Eg 2: Let AL = 6A.

As lower nibble (A) is >9 then add 06 ⇒ A+06



18 AAS: ASCII Adjust AL after subtraction

AAS instruction corrects the results in AL register after subtracting two ASCII operands. The result is in unpacked decimal format.

The procedure is similar to AAA except for the subtraction of 06 from AL.

If lower nibble of AL register is greater than 9 or if AF = 1

then AL = AL − 6 &

AH = AH − 1

CF & AF = 1

otherwise the result is needs not correction.

19: AAM: ASCII Adjust after Multiplication:

It converts the product present in AL into unpacked BCD format

Eg: Assume AL_0 = 04 ; BL = 09 when AL × BL is multiplied using MUL the result will be stored in AX which will be equal to 24H (36D).

After AAM is used the AX will contain 03 & 06 in AH & AL respectively which are nothing but decimal values.

20) AAD: ASCII Adjust before Division:

AAD converts two unpacked BCD digits in AH & AL to the Equivalent Binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD Byte.

LOGICAL Instructions:

1) AND: Logical AND:

This Instruction performs bit wise AND operation of Source & destination operand and stores the result in destination operand.

The Source can be either Immediate/Register/Memory.
The destination can be either Register/Memory
Both Source & destination cannot be Memory location at once.

Eg: 1) AND AX, 000E H.
2) AND AX, BX

2) OR: Logical OR:

The OR instructions also performs the bit wise OR operation of Source & destination operand & stores the result in destination operand.
The Source & destination rules for will be same as AND operation.

Eg: 1) OR AX, 0027 H.
2) OR AX, BX

3) Not: logical Invert:

The Not instruction complements the contents of an operand register or Memory location bit by bit.

4. XOR: Logical Exclusive OR:

XOR performs bit wise Exclusive OR operation. The constraints on the operands also remain the same. The XOR operation gives a high output when the 2 input bits are dissimilar. Otherwise the output is zero.

Eg: 1) XOR AX, 0098H.

2) XOR AX, BX

3) XOR AX, [5000H]

5) Test: Logical Compare instruction: The Test instruction performs a bit by bit logical AND operation on the two operational data. Each bit is set to 1 if the bits of the operands is 1, else it is zero. The result of this instruction is not available in the destination it will be available only in the flag registers ie Only the flags are affected. The operands may be registers, memory or immediate data

Eg: 1) Test AX, BX

2) Test [0500], 06H

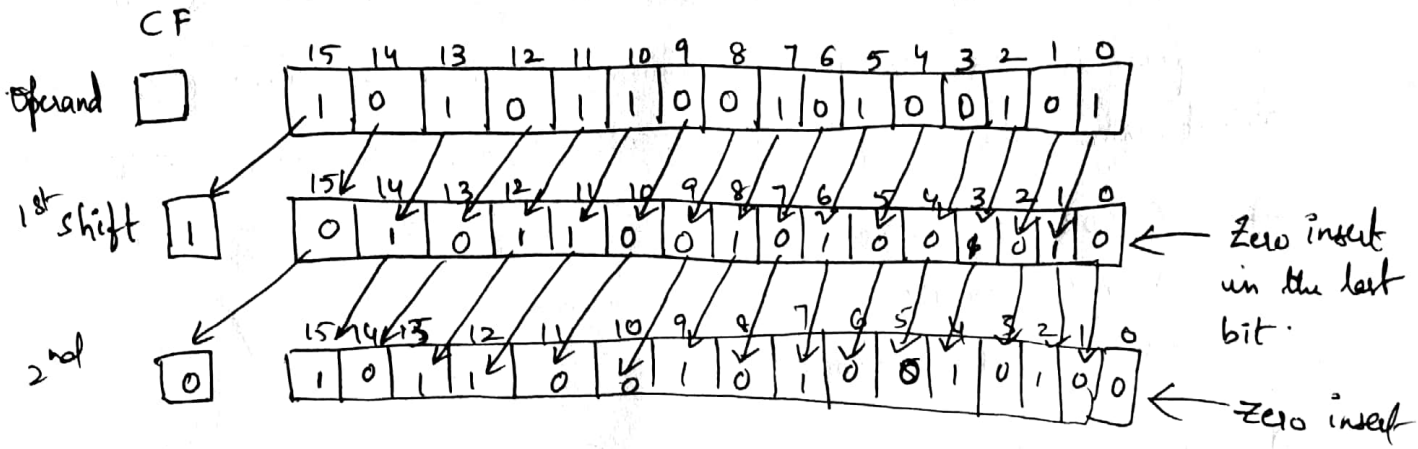
3) TEST BX, CX

6. SHL/SAL: Shift Logical/Arithmetic left:

It shifts the operand bit by bit to the left and Insert Zeros in the newly introduced least significant bits. In Case of Shift & rotate instructions, the count is either 1 or Specified by Register CL. The data Cannot be an immediate ~~flag~~ data.

The Shift operation is through Carry flag.

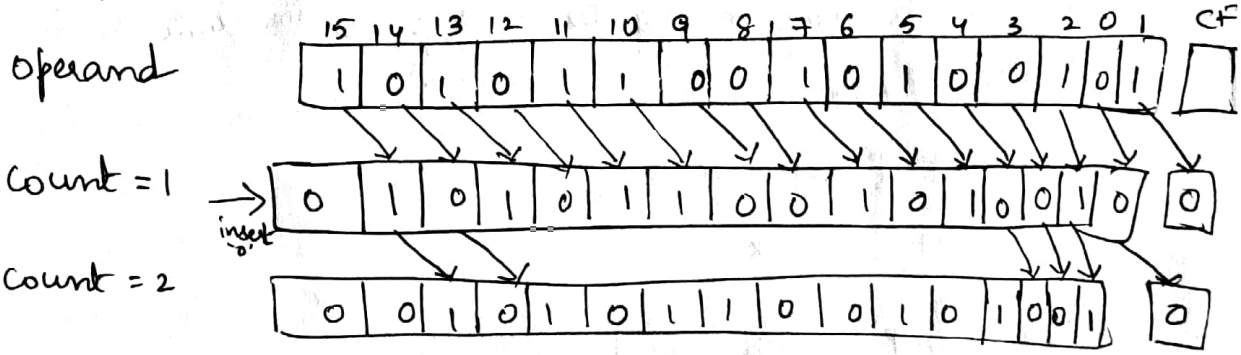
If the Operand = ACA5



#) SHR: Shift logical Right: This instruction performs bit-wise right shifts on operand by specified Count and inserts Zero in the shifted Positions.

(15 - 0) ⟶ & CF

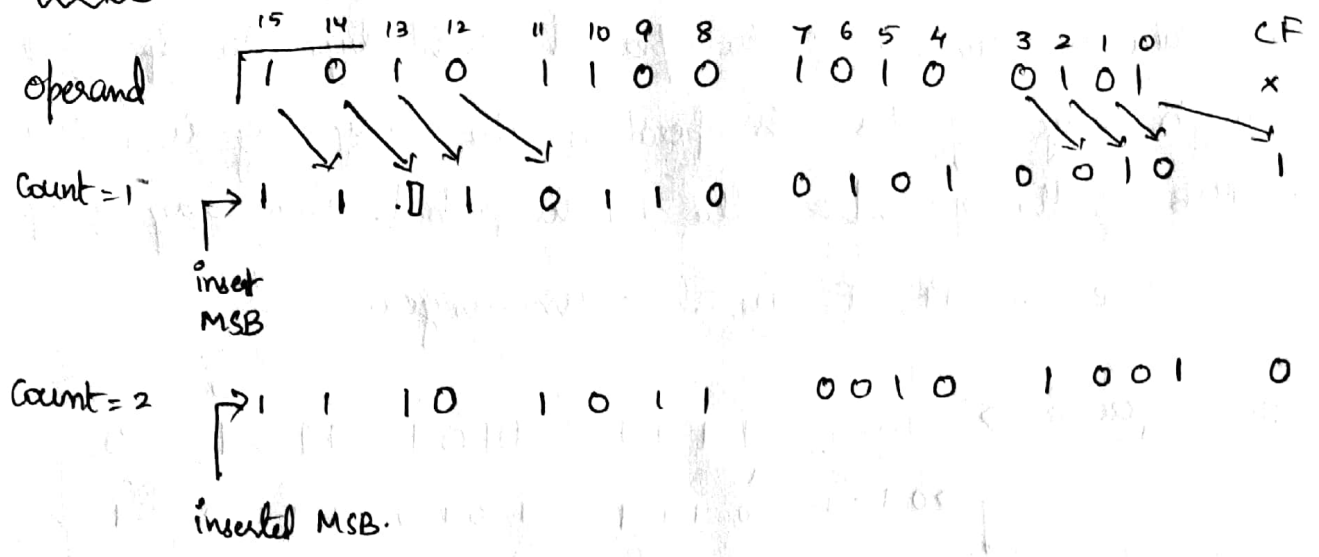
Eg: Bit positions 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1...



8) SAR; Shift Arithmetic Right:

It performs right shift on operand, that may be a register or a memory location. by specified Count in the instruction. It inserts the Most Significant bit of the operand in the newly inserted positions. All the Conditional Code flags are affected.

Example:

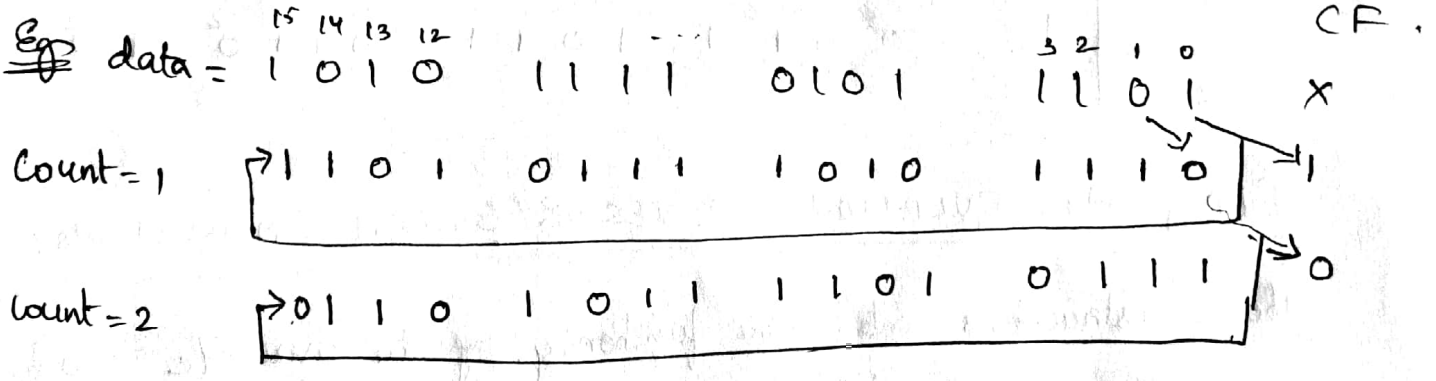


9) ROR: Rotate Right without Carry:

The instruction rotates the contents of the destination operand to the right either by one or by the count specified in CL, excluding Carry.

The LSB is pushed into the Carry flag & simultaneously it is transferred into MSB

The PF, SF, & ZF are left unchanged.



10) ROL: Rotate left without Carry:

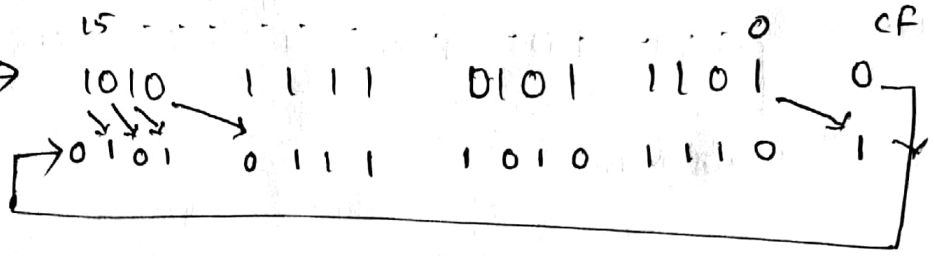
The instruction rotates the content of the destination operand to the left by the specified count excluding Carry. The MSB will is pushed into the Carry flag as well as the least significant bit at each operation. Other than this operation everything remains the same.

11. RCR: Rotate Right through Carry

It rotates the contents of the operand right by the specified count through Carry flag. For each operation, the Carry flag is pushed into the MSB of the operand & the LSB is pushed into Carry flag.

The SF, PF, ZF are left unchanged.

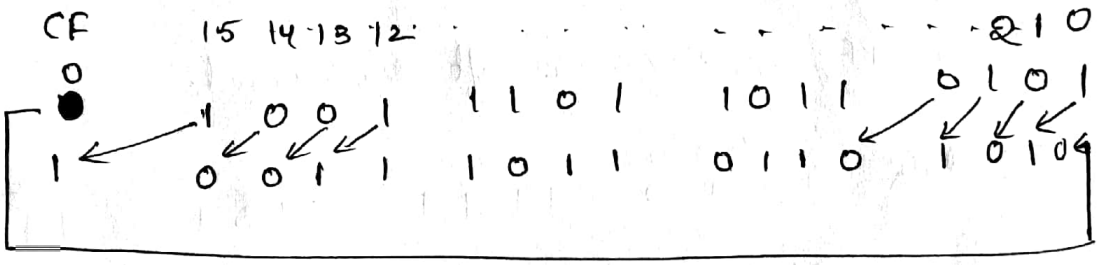
Eg



12 RCL: Rotate Left through Carry.

It rotates the contents of the destination operand left specified by the specified count through the Carry flag (CF). The entire operation will remain the same as in RCR.

Eg



FLAG MANIPULATION & PROCESSOR CONTROL INSTRUCTIONS:

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

1) flag manipulation.
2) Machine Control

The flag manipulation instructions directly modify some of the Carry (CF), Direction Flag (DF) & Interrupt Flag (IF) flags directly.
In a content, the IF & DF are also called as Machine Control

flags. The all the other flags can be Modified by POPF & SAHF LAHF etc.

The Machine control flags are as follows and they do not require any operand

Flag manipulation Instructions

1) clc — Clear CF ⇒ CF ← 0
2) CMC — Complement Carry Flag ⇒ CF ← $\overline{CF}$
3) STC — Set Carry Flag CF ← 1
4) CLD — Clear DF (Direction flag) ← 0
5) STD — Set DF ← 1
6) CLI — clear Interrupt flag IF ← 0
7) STI — set Interrupt flag IF ← 1

Machine Control Instructions:

1) Wait – wait for Test Input pin to go low
2) HLT – Halt the processor after Interrupt is Set.
3) NOP → No operation.
4) ESC → used to pass Instruction to a Co-processor which shares the address & data bus with 8086
5) LOCK → Lock the Bus used with Instruction prefix LOCK.

BRANCH INSTRUCTIONS:

The control transfer Instructions transfer the flow of Execution of the Program to a new address Specified in the instruction directly or indirectly. These Branch instructions are classified into two types. They are

1) unconditional Branch Instructions
2) Conditional Branch Instructions.

1) unconditional Branch Instructions:

1. CALL: This Instruction is used to Call a Subroutine from a main Program. The address may be Specified directly or indirectly depending on addressing mode.

2. RET: This Instruction is used to return from a procedure. At Each CALL Instruction, the IP and CS of the next Instruction is pushed onto stack, which before the Control is transferred to the procedure. At the End of the procedure, RET Instruction must be Executed. It takes & reloads the value of CS & IP from the Stack.

3. INT N: This Instruction Executes the type of instruction based on the number N. The number N is multiplied by 4 and the control will be transferred to the location obtained from the location after multiplicatio

4) JMP: This Instruction unconditionally transfers the Control of Execution to the Specified address using an 8 bit or 16 bit displacem

5) IRET: IRET is executed or used to return from the Interrupt Service routine.

6) Loop: Loop unconditionally: This instruction Executes the part of the program from the Label or address Specified in the instruction up to the loop Instruction, CX number of times. At each iteration, CX is decremented automatically.

2) Conditional Branch Instruction:

When these instructions are executed, execution control is transferred to the address specified relatively in the instruction, provided the condition implicit in the opcode is satisfied. If not the execution continues sequentially.

The different conditional branch instructions are as follows.

1.	JZ / JE	Jump if zero / Equal	Transfers the Execution to a label if ZF = 1
2.	JNZ / JE	Jump if Not zero / equal	Transfers the Execution to a label if Z = 0
3.	JS	Jump if SF is set	Transfers the Execution to a label if S = 1
4	JNS	Jump if SF is cleared	Transfers the Execution to a label if S = 0
5	JO	Jump if overflow	Transfers the Execution to a label if OF = 1
6	JNO	Jump if no overflow	Transfers the Execution to a label if OF = 0
7	JP / JPE	Jump if even papity	Transfers Execution if PF = 1
8	JNP	Jump if not even parity	Transfers the Execution if PF = 0
9.	JB / JNAE / JC	Jump if Below / Jump if carry / Jump if not above	Transfers the execution if CF = 1
10	JNB / JAE / JNC	Jump if not Below / Jump if Above or Equal / Jump if No carry	Transfers the Execution if CF = 0
11.	JBE / JNA	Jump if Below Equal / Jump if not above	Transfers the Execution if CF = 1 OR ZF = 1
12	JNBE / JA	Jump if not Below Equal / Jump if above	Transfers the Execution if CF = 0 ZF = 0

STRING MANIPULATION INSTRUCTIONS:

A series of data bytes or words available in memory at consecutive memory locations, called as byte strings or word strings.

While refering to a string, the parameters required are

1) starting and End address of string
2) length of the string.

The length of the string is usually stored in a count register usually CX and it has to be updated on each iteration. In some the increment or decrement of the counter/pointer is dependent on the Direction flag.

If DF = 1 then it is Auto decrement mode

If DF = 0 then it is Auto increment mode

If the operation is on a byte string then the pointer will be incremented ~~and~~ or decremented by 1

If the operation is on a byte word then the pointer will be decremented or incremented by 1

Instructions:

The different types of string instructions are

1) MOVSB/MOVSW :
2) CMPSB/CMPSW
3) SCASB/SCASW
4) LODSB/LODSBW
5) STOSB/STOSW
6) REP – This instruction is used along with other instructions as mentioned above

1. MOVSB/MOVSW : Move string Byte or String word:

This instruction is used move a set of memory locations from Source to destination location. The Source address is given by DS:SI and the destination's starting address is given by ES:DI and the Physical address is given by 10H * Ds + [SI] {Source address} & 10H * Es + [DI] {destination address} respectively.

The REP instruction prefix is used with Movs Instruction to repeat it by a given value in Cx (Counter). No flags are affected in this instruction.

Eg:

1) MOVSB	Move string from DS:SI → ES:DI ; SI ← SI ± 1 ; DI ← DI ± 1
2) MOVSW	Move string from DS:SI → ES:DI ; SI ← SI ± 2 DI ← DI ± 2

2. CMPS: Compare string Byte or string word:

The CMPS instruction can be used to compare two strings of bytes or words The length of the string must be stored in string register Cx. If both byte or word strings are Equal, Zero flag is set. The DS:SI and ES:DI point to two strings. The REP instructions prefix is used to repeat the operation till Cx (counter) becomes Zero or the condition specified by REP becomes false

The Comparision Starts from the initial address and after each comparision the index registers are updated depending on the direction flags and then the Counter is updated. When there is a Mismatch, the Carry or Zero flags are updated and the next instruction is executed.

Eg 1) CMPSB ; Compares two string Bytes (DS) + (SI) - (ES) + (DI)

SI ← SI ± 1

DI ← DI ± 1

2) CMPSW ; Compares two string words (DS) + (SI) - (ES) + (DI)

SI ← SI ± 2

DI ← DI ± 2

3. SCAS : 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 by ES:DI register pair. The length of the string is stored in CX. Whenever an operand specified in AL or AX is found in the string, Execution stops and the Zero flag is set. If no match is found Zero flag is set. The pointers and counters are updated automatically till the match is found.

Eg 1) SCASB ; Scans for an operand stored in AL or AX (Searches)

; Ax - ES:DI

DI ← DI ± 1 or 2

4) LODS : Load String Byte or string word:

The lods instruction loads the AL/Ax register by the content of the string pointed by DS:SI register pair. The SI is modified automatically depending upon DF. If it is a Byte transfer (LODSB), the SI is modified by one and if it is a word transfer (LODSW), the SI is modified by two. No other flags are affected by this instruction.

Eg 1) LODSB ; load String in DS:SI → AX

AL ← (Ds) + (SI) → Ax

SI ← (SI) ± 1

2) LODSW ; load String in Ds:SI → AX

SI ← SI ± 2

5) STOS: 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.

Eg 1) STOSB ; Copies the operand present in AL/Ax register to the location pointed by ES:DI.

Ax → (ES) + (DI)

(SI) ← (SI) ± 1

2) STOSW ; Copies the operand in AL/Ax register to destinatio

Ax → (ES) + (DI)

(SI) ← SI ± 2

REP: Repeat Instruction Prefix: This Instruction is used as a Prefix to other instructions. REP is used such that the Execution is repeated until the Cx register becomes Zero. (Automatically). When Cx becomes zero, the Execution proceeds to the next Instruction in Sequence.

The other forms of REP that can be used with CMPS & SCAS are

1) REPNE/REPNZ : Repeat if not Equal / Repeat if not Zero

2) REPE/REPZ : Repeat if Equal / Repeat if Zero

ASSEMBLER DIRECTIVES :

An Assembler converts the assembly language program into its Equivalent machine code modules. The Assembler also debugs the program and generate syntax errors. but the logical errors are not generated by Assembler. The logical errors such as required Storage, Constant or variable type, Logical names of Segments etc must be directed by the programmer only. Such directions are Assembler directives.

The predefined alphabetical strings which are used to define the various logical elements of the processor are called as assembler directives. These are also called as pseudo instructions.

Eg: DB, DW, DQ, DT, ASSUME, END, SEGMENT, PROC, Etc.

1) DB : Define Byte :

The DB directive is used to reserve byte or bytes of memory locations in the available memory.

The range is given by 00H – FFH for unsigned value,
00H – 7FH for positive value
80H – FFH for negative value.

Syntax : Variable_name DB Value/value.

Example : LIST DB 7FH, 42H, 35H.

Three Consecutive memory are reserved with the variable name LIST and the values are 7FH, 42H, 35H

2) DW : Define word

The DW directive is used to reserve a word of consecutive memory location in the available memory. 1 word is equals to 2 bytes. Two

The range of DW is given as 0000 - $FFFF_H$ for unsigned values
0000 - 7FFFH for positive values
8000 - FFFFH for negative value.

Syntax : Variable name DW value/values.

Example : LIST DW 6512H, 0F25H, 0CDEH

3) ASSUME : Assume Logical Segment Name :

The Assume directive is used to inform the assembler, the names of the logical Segments to be assumed for different Segments used in the Program.

Eg : The Code Segment is given the name CODE, data Segment may be given DATA etc.

The statement ASSUME CS: CODE directs the Assembler that the machine codes are available in a Segment name CODE & hence the CS register is to be loaded with Segment address allotted by the Operating System for the label CODE.

4) END : END of program :

The END directive marks the end of an assembly language program. When the Assembler comes across the END directive, it ignores the Source code after END and hence it is usually the last line of the program.

5) ENDP : End of procedure :

The Subroutines are called as procedures in 8086. The ENDP is the directive which is used to indicate the end of the procedure. A procedure is assigned a Label always and the end of the

Procedure should also indicate the name of the procedure.

Eg: Assume that a subroutine to calculate factorial is defined then the part of the program is shown below

```
PROC   FACTORIAL
  :
  :   } subroutine code.
  :
FACTORIAL ENDP.
```

6) ENDS : END OF SEGMENT

This directive marks the end of a logical statement. The logical segments are assigned with the names using ASSUME. The names appear with the ENDS directive to mark the end of those particular segments

Eg:

```
DATA   SEGMENT
  :
  :
DATA   ENDS
ASSUME   CS: CODE , DS: DATA
CODE   SEGMENT
  :
  :
CODE   ENDS.
```

7) EVEN : ALIGN ON EVEN Memory address :

The Assembler generally assigns the address sequentially to all the elements in the source code.

The EVEN directive updates the location counter to the next even address, if the current location counter is not even. When the assembler comes across the EVEN directive it checks the current address to be allocated to the next Element. If this current address is EVEN then that address is allocated else that address will not be updated.

9) EQU: EQUATE:

The directive EQU is used to assign a label with a value or a Symbol The use of this directive is just to reduce the recurrence of the numerical values or constant in a program code. The recurring value is assigned with a label and this label will be used throughout the Program.

10) ORG: ORIGIN:

The ORG Directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the statement.

If ORG is not written in the program then the assembler assigns the code from 0000 H.

If ORG 200 H is written in the program then the code segment will be from 200 H

11) PROC: Procedure: 12) NEAR: 13) FAR:

The PROC directive marks the start of a named procedure in the statement. Also the types Near & FAR specify the type of the procedure

NEAR indicates intersegment call
FAR indicates intrasegment CALL

14) SEGMENT: LOGICAL SEGMENT:

The segment directive indicates the start of a logical segment. These logical segments must be assigned a name by this statement

Eg: CODE SEGMENT → This lines defines the start of the Code Segment. The word CODE is the name of the Segment that has to be initiated.

15 SHORT:

The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump i.e it must be within -128 to +127 bytes (from the address of the byte next to the Jump opcode). This method of specifying the jump saves the memory

Eg: JMP SHORT LABEL.

MACROS: